The SIS Project: Software Reuse with a Natural Language Approach

Lutz Prechelt (prechelt@ira.uka.de)
Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe
Postfach 6980
D–7500 Karlsruhe, Germany

Technical Report 2/92

June 16, 1992

Abstract

The SIS (Software Information System) project investigated a new approach to one part of the software reuse problem. The problem is how to find and use a reusable component from a repository. The approach is (1) to provide a knowledge representation system that allows for modeling the components in the repository with user-defined semantic categories, (2) to provide deductive capabilities in this knowledge representation system, to perform powerful retrieval operations, and (3) to complement the formal query facility with a natural language user interface in order to achieve ease of use and freedom from terminology constraints imposed by the names used in the knowledge base.

In the SIS project, both the knowledge representation system YAKS and the natural language processing component SARA have been designed and built. We have developed a general approach for building transportable natural language interfaces for databases using KL-ONE-like knowledge representation languages. A prototype system called YAKR was built for which this approach was used to integrate YAKS and SARA. This report describes the philosophy, design, and implementation of YAKR, the first experiences we have gathered with it, and the limitations we expect.

Contents

1 Introduction **4**
1.1 Underlying Assumptions . . . 4
1.2 Requirements . . . 4
1.3 Basic Design . . . 5

2 The YAKS Knowledge Representation System **6**

3 YAKS Knowledge Acquisition **8**
3.1 Our Experimental Modeling . . . 9
3.2 A Knowledge Acquisition Tool . . . 9

4 The SARA Natural Language Processor **10**

5 SARA Knowledge Acquisition **11**
5.1 Information about Individual Words . . . 13
5.2 Information about Cases . . . 14
5.3 Information about Explicit Inheritance of Words . . . 15

6 Integration of YAKS and SARA **15**
6.1 Generation of Case Frames . . . 15
6.1.1 Action-or-Object Concepts . . . 16
6.1.2 Attribute Concepts . . . 16
6.1.3 Has-Action-or-Object Roles . . . 17
6.1.4 Has-Attribute Roles . . . 17
6.1.5 Theme Roles . . . 18
6.1.6 Synonymless Roles . . . 18
6.2 Generation of Queries . . . 18

7 Experiences and Limitations **19**
7.1 Usefulness . . . 19
7.2 Practicability . . . 20
7.3 Acceptance . . . 20
7.4 Scalability . . . 20
7.5 Applicability . . . 20

8 Related Work **21**
8.1 Finding Reusable Components . . . 21
8.2 Transportable Natural Language Interfaces . . . 22

9 Results **23**

10 Further Work **24**

A Other knowledge sources of YAKR **25**
A.1 SARA dictionary . . . 25
A.1.1 Verbs . . . 25
A.1.2 Nouns . . . 26
A.1.3 Adjectives . . . 27
A.2 SARA role table . . . 27

B List of Generated Case Frames **28**

C User interface commands **30**

D Example Session **31**

E Possible Problems with Natural Language Interfaces **32**

Bibliography **36**

1 Introduction

The software component reuse problem can be split into two parts: (1) how to create and collect reusable components and (2) how to actually reuse them. The SIS project was only concerned with the second part.

The problem of how to reuse components can be further split into three parts: (a) how to find a suitable component for a given problem, (b) how to adapt it for the problem, if necessary, and (c) how to actually use it correctly. The SIS project attempted to build a system that mainly addresses (a), the other points are only partly covered. This system is called YAKR.

The following subsections describe the assumptions YAKR is based on, the requirements that follow from those assumptions and how these requirements are shaped into a concrete system design.

1.1 Underlying Assumptions

There are a few assumptions that underly the design of YAKR:

A1: When a user starts looking for a reusable software component for a certain task, often no complete or exact specification for the functionality of that component is available; the user's conception of what is needed is fuzzy.

A2: It is difficult to describe a component needed for a task X in a terminology that is not from the domain of X.

A3: Programmers tend to rewrite software components instead of reusing existing ones, unless reuse is easy and fast.

A4: It is unlikely that a repository will be maintained with enough care in the long run, if the database that is used to search the repository must be kept consistent with a complicated schema.

These or similar hypotheses are supported by observations of many scientists in the area of software engineering (see for example [6, 29, 35, 40, 18, 22]), although especially A4 is often not addressed at all in papers about concrete software reuse support systems.

1.2 Requirements

From these assumptions we derive one-to-one the following requirements as guidelines for the design of YAKR:

R1: The system must allow successful querying with partial and inaccurate specifications.

R2: The system must allow many alternative ways to express the same request.

R3: The user interface must be as simple as possible.

R4: The internal representation formalism for the software components must be easy to use.

There are probably several ways to fulfill these requirements. In the SIS project, we picked one of them and showed how to implement and operate it. The design of our system is outlined in the following subsection.

1.3 Basic Design

R1 is realized with a specially designed knowledge representation language, YAKS (Yet Another Knowledge representation System[1]), which has similarities to KL-ONE [8]. The constructs of the language are directly targeted to the description of software components and allow to define suitable terminology for software from any domain. This terminology is arranged in a taxonomy, which allows incomplete as well as inaccurate queries to be answered: they just retrieve elements that are more general in respect to the taxonomy.

R2 is satisfied by a natural language interface. Natural language is the most versatile way of expression. Usually, natural language interfaces have the disadvantage that they are expensive to construct and to transport to a new domain. We have found a way to minimize the work that is needed to construct or transport the interface: only short annotations to each definition in the terminology are needed.

R3 is trivially solved by the natural interface, too. Since natural language is the easiest way for a human to express a specification, especially if its conception is still fuzzy, the tendency not to use the software repository is minimized.

R4 actually means to solve two maintenance problems in our system: First, it must be easy to add descriptions of new software components. This ideal is approached by letting the knowledge representation be *hybrid*, consisting of a terminological part and an assertional part. The terminological part is used to model the semantic categories and relations of a domain, but does not state any specific facts about the repository. The assertional part describes the actual objects in the repository in terms of what was defined in the terminology. So the usual case when adding new software components is just to describe them in the compact, adequate and natural terms of the terminology that has once been defined before. Only if the components belong to a part of the domain that has not yet been adequately modeled in the terminology, the work to add these components is complicated. Thus, although we use a complex knowledge representation, maintenance of the knowledge base is easy; the difficult part of the work is concentrated on the initial creation of the knowledge base for a repository.

Second, it must be easy to add the natural language information for new software components. This requirement is fulfilled prototypically by our design: If no change in the taxonomy is needed, only a name needs to be given for any new component. If the taxonomy is refined, the new entries in it have to be annotated, as mentioned above. Thus, this aspect of maintenance is also easy to master.

The reduction of natural language information to small amounts of annotation is achieved with a special architecture that tightly couples the natural language parser SARA and the knowledge representation system YAKS (see figure 1): From the annotations, all information that the parser needs can be generated automatically by a so-called case frame generator. The output of the parser then contains enough information about the structure of the knowledge base to allow a relatively simple recursive process be sufficient to transform parser output into knowledge base queries. This is done by the so-called case frame interpreter. This coupling avoids the need for complicated grammar or lexicon writing and is an especially important aspect of YAKR's design.

The rest of this report is organized as follows: The sections 2 to 5 describe the YAKS knowledge representation system, the SARA natural language processing component, and the knowledge acquisition processes for both. Section 6 describes how SARA and YAKS are integrated into a system that satisfies R1–R4. Section 7 describes first experiences with our system and discusses the limitations of both our current system and the approach in general. Section 8 compares YAKR to related work in finding reusable software components and in building transportable natural language interfaces. A

[1] In fact, the language is called YAKL (for obvious reasons); only the program that implements it is called YAKS. We will ignore that difference in this report.

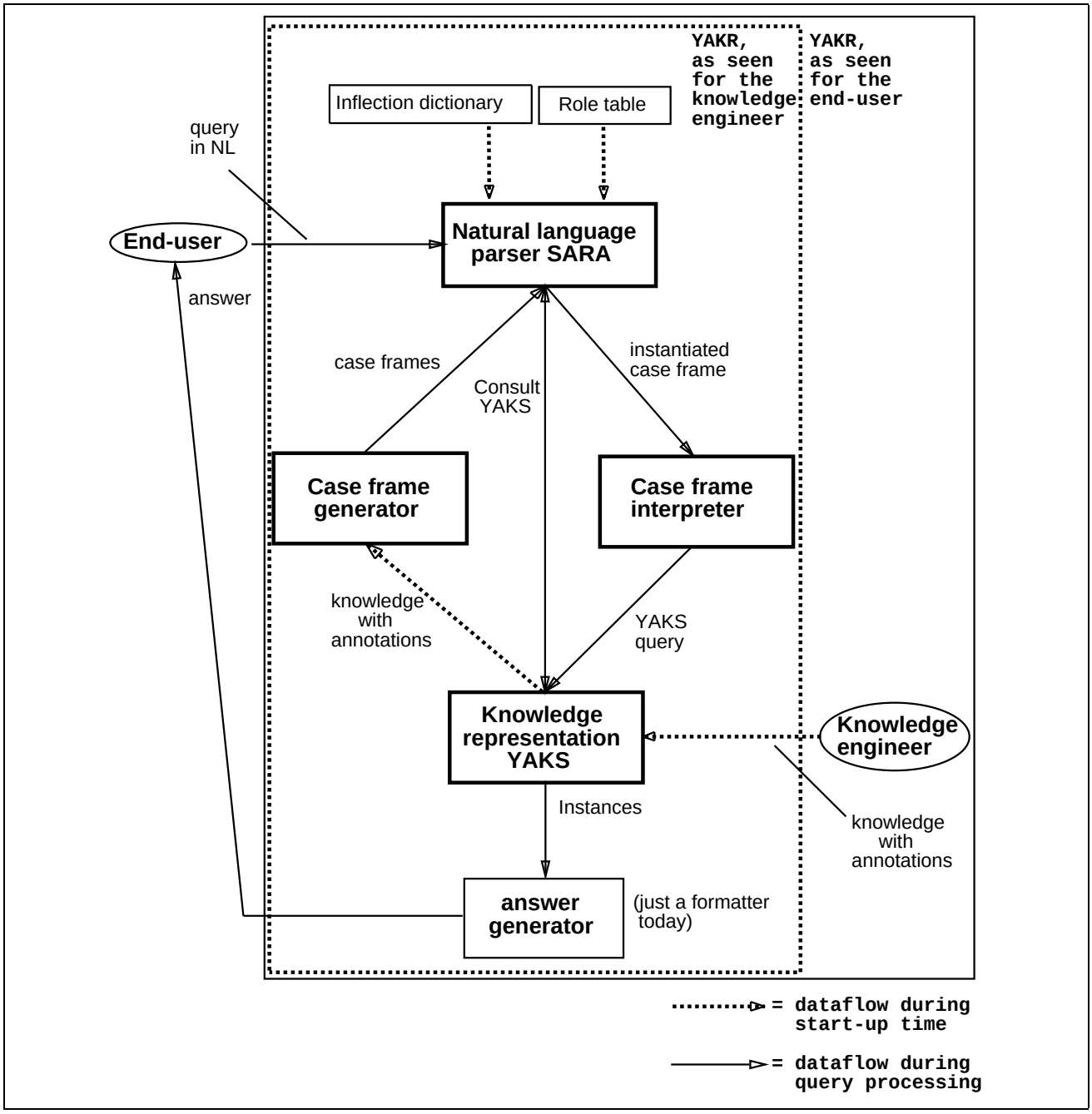

Figure 1: Basic architecture and dataflow of YAKR

section discussing related work and a section that summarizes our results follow. Several appendices provide additional details.

2 The YAKS Knowledge Representation System

The YAKS knowledge representation language (in a former version named KRS [1, 17]) has well defined model-theoretic semantics and distinguishes between assertional and terminological knowledge. The *terminological knowledge* defines a "vocabulary" to be used to express facts. The *assertional knowledge*

comprises facts about *individuals* in the application domain.

The terminological knowledge consists of concept definitions and role definitions. A *concept* can be thought of as an abstract set of individuals. The concrete individuals that belong to a concept are called the *instances* of that concept. A *role* is a binary relation from a concept A to a concept B, i.e., a set of pairs of instances. A is called the *domain* of the role and B is called the *range* of the role.

Concepts are defined with constructors that each describe a subset of the set of all possible individuals. Each constructor thus represents a restriction that an instance must adhere to in order to belong to the concept that is defined by the constructor. Roles are defined with constructors, too. There is a distinction between primitive concepts or roles (partial descriptions, describing conditions that are necessary), defined concepts or roles (exact descriptions, describing conditions that are both necessary and sufficient), and derived concepts or roles (describing conditions that are sufficient, but not necessary).

The concept language in YAKS is quite expressive, allowing value restriction, number restriction for role fillers, role–value maps, conjunction, disjunction, and negation. The concepts form a *hierarchy (taxonomy)* with multiple inheritance. The role language allows conjunction, disjunction, domain restriction, range restriction, negation, and inversion.

As a simple example, look at the following definitions, excerpted from the larger example on page 11:

```
DEF-CONCEPT Input-Functions = AND(Functions  SOME(reads)).
       ROLES(reads : Data-Objects;).

PRIM-CONCEPT Data-Objects = Objects.
```

These definitions mean the following:

1. *Input-Functions* is a defined concept, i.e. exactly all those instances that obey the restrictions given in the concept definition belong to the concept.

2. there are two restrictions, that instances must obey, in order to belong to the concept *Input-Functions*: (a) they must belong to the concept *Functions* and (b) they must have at least one value assigned for the role *reads*.

3. *reads* is a role, whose domain contains at least *Input-Functions* and whose range contains at least *Data-Objects*. Additional declarations of the role at other concept definitions may modify this domain and range.

4. *Data-Objects* is a primitive concept, i.e. only those instances that obey the restrictions given in the concept definition may belong to the concept, but we can not know whether they really belong to it unless we are explicitly told so.

5. the only restriction of *Data-Objects* is that they must be *Objects* (whose definition is not shown here).

There are a number of inferences which can be made on concepts and roles. The most important ones are *subsumption, classification, realisation*, and *retrieval*. Subsumption determines whether a given concept A subsumes a second concept B, e.g., whether in all possible world descriptions all individuals that belong to B also belong to A. Classification determines the correct placement of a new concept in a given concept hierarchy, that is, which concepts are the immediate superconcepts and the immediate subconcepts. Realisation determines for an individual the most specific concepts

it belongs to. Finally, retrieval determines for a given concept description the set of all individuals that are instances of the concept. Retrieval is analogously defined for roles, producing a set of pairs of instances. The resulting taxonomy after classification of all concepts and realisation of all instances for the larger example on page 11 is shown in figure 2 on page 13. Since the YAKS language is so powerful, the inferences are not completely computable. Yet, we have not found a single case where this has been a problem in practice.

As a simple deduction example, look at the following definition, also excerpted from the larger example on page 11:

```
INSTANCE fgetc = AND(Functions
                     VALUES(reads, [character-c])
                     VALUES(has-Parameter, [filepointer-stream])
                     VALUES(has-synonym, ["fgetc"])).
```

This definition means the following:

1. *fgetc* is an instance (individual).
2. *fgetc* belongs to the concept *Functions*.
3. *fgetc* has the instance *character-c* (whose definition is not shown, but which belongs to the concept *Data-Objects*) as filler of the role *reads*.
4. *fgetc* has the instance *filepointer-stream* (whose definition is not shown, but which belongs to the concept *Data-Objects*) as filler of the role *has-Parameter*.
5. *fgetc* has the string instance *"fgetc"* (which need not be defined, because it is a string) as filler of the role *has-synonym*.

Points 1–3 of above allow YAKS to infer that *fgetc* belongs not only to *Functions* but also to *Input-Functions* because it adheres to all both restrictions given in the definition of *Input-Functions* and *Input-Functions* is a defined concept (i.e. any instance that adheres to all restrictions given in the concept definition is guaranteed to belong to the concept).

YAKS has a uniform language for data modeling (describing concepts and roles) and querying, i.e., all constructs that can be used to model a concept or role can also be used to query for one. Thus, the query language is quite powerful and makes YAKS more expressive than a relational database. This power is especially important for the natural language interface, because the query language limits which kind of natural language terms and phrases may be interpreted and how they can be interpreted. In YAKS, all limitations of the query language stem from limitations of the modeling language.

For a detailed description of YAKS see [2].

3 YAKS Knowledge Acquisition

In principle, the knowledge acquisition process for YAKS has two main stages: First, define the terminology that will be used to describe the domain that is to be modeled. Second, create the actual contents of the database by defining the instances of the domain in terms of the terminology that has been defined.

This process resembles object-oriented design of a software system: First the classes (concepts) have to be described, i.e., "find out which kinds of objects exist and which of them are special cases of which others". The better this modeling is, the easier the second stage will be: Define all the actual objects (instances) by picking a class (concept) for each of them and instantiating all its attributes (assigning its role fillers).

In practice, just as in object-oriented design, some backtracking will usually be necessary in order to get the terminology right. Our experience indicates that modeling in YAKS is about as difficult as class design in an object-oriented programming language: If the task is complicated, modeling is a challenging task. But once the modeling is right, everything looks simple and clear.

A tiny example of what a YAKS modelling may look like will be given below in section 5.

3.1 Our Experimental Modeling

To learn about how modeling actually works and how good our system behaves on a medium-sized example, we have modeled a part of the internal view of the NIH Class Library [19], which is written in C++. The most complicated part of this task was to model the constructs of the C++ programming language. The resulting terminology contains about 160 concepts and 130 roles in 40 Kilobytes of YAKS source code. The actual knowledge base for NIH-CL models only the top three classes of the NIH-CL class hierarchy in full detail (and the rest very roughly), but nevertheless contains almost 1000 instances and some dozens of help concepts in 105 Kilobytes of YAKS source code.

It took almost 400 person hours to create this model, although it must be mentioned that YAKS was still under development in this period, i.e. unreliable as well as changing. We also first had to learn how to model properly. Furthermore, C++ with all its exceptions and design flaws turned out to be an awfully complicated domain.

3.2 A Knowledge Acquisition Tool

When modeling the internal view of software components, the biggest part of the instance descriptions can be built automatically, because it consists of information about declarations and cross references. We have in fact built a tool for this purpose, called `CPPREF`, to be used on C++ programs. It generates instance descriptions about classes and functions, based on our modeling of C++. Only the purpose of classes and functions still has to be modeled by hand, because it is not extractable from the source code of a program.

In fact, the amount of data that is produced by this tool is overwhelming: Out of 44 header files (140 Kilobytes) from the NIH library, `CPPREF` created 1 Megabyte of YAKS output; for the implementation files, the ratio is a bit better. Thus it is unrealistic even for programs of moderate size, to feed *all* the internal interactions into YAKS. Fortunately, it is easy to reduce the amounts of information drastically by excluding certain types of information from the output of `CPPREF`.

To give an idea of what information is computed by `CPPREF`, we give a list of all relevant roles here: begins-in-line, calls, contains, declares, declares-access-private, declares-access-protected, declares-private, declares-protected, declares-public, defines, defines-private, defines-protected, defines-public, ends-in-line, exports, has-access-mode, has-base-class, has-base-file, has-basetype, has-cast, has-class-type, has-constructor, has-creation-date, has-datatype, has-default, has-derived-class, has-descendant-class, has-destructor, has-dimension, has-directory, has-enumerator, has-friend, has-function-call, has-implementation, has-initialization, has-inner-block, has-length, has-linkage, has-linkagetype, has-local-variables, has-member, has-module, has-name, has-number, has-outer-block, has-owner, has-parameter, has-private-base-class, has-private-member, has-protected-member, has-public-base-class, has-public-member, has-purpose, has-returntype, has-scope, has-size,

has-specification, has-specifier, has-subclass, has-superclass, has-synonym, has-virtual-base-class, imports, includes, inherits, is-datatype-of, is-declared-in, is-defined-in, is-enumerator-of, is-exported-by, is-exported-to, is-friend-of, is-included-by, is-inherited-by, is-local-class-from, is-member-of, is-parameter-of, is-private-base-class-of, is-private-member-of, is-protected-member-of, is-public-base-class-of, is-public-member-of, is-used-by, is-virtual-base-class-of, of, reads, redefines, uses, with, writes.

These roles are only partially computed by `CPPREF` itself, most of them are deduced from others by YAKS. For detailed information about this tool, see [34]

4 The SARA Natural Language Processor

The effort to implement a sophisticated natural language processor is high. Therefore, we chose to implement a case frame parser [38, 12], which is based on the notion of semantic case [16] and requires only moderate implementation effort compared to other parsing techniques, as for example unification-based parsing [33]. Case frame parsing provides a good starting point for building robust parsers and for handling dialogue phenomena such as ellipsis [10, 13, 11], since least-error fitting of case frames can be used. The main disadvantage of case frame parsing is that complex syntactical constructions cannot be handled adequately.

However, the syntactic coverage of our case frame parser SARA [24], which targets German, is quite broad compared to other parsers using this technology; it handles the following syntactic phenomena: Noun phrases with determiners, adjectives, adverbs, restricted numbering, and simple quantifiers; simple conjunctions and disjunctions of noun phrases; all types of sentences (Yes/no–questions, Wh–questions, declaratives, imperatives, infinitives, to-be–sentences, to-have–sentences) in all tenses; modal verbs, immediate relative clauses, simple sentence-level negations, meaningless words, and subclauses starting with a conjunction (e.g. ...,*because*...). Not handled are general and/or–conjunctions, general numbers, general quantifiers, and many complex syntactical constructions.

Case frame parsers convert written input into instantiated case frames as an intermediate representation; no surface structure is generated. The parser uses built-in syntactic knowledge and a set of case frames, representing semantic knowledge. Thus, the parsing is semantically directed. A case frame represents an utterance by its central concept, called the *head* of the case frame, plus a set of *cases*, each of which describes (a) a certain semantic relation to the head and (b) what may appear in that relation (the *fillers*). There are verbal case frames describing sentences (with a main verb as head) and nominal case frames describing noun phrases (with a noun as head). Each case frame represents a whole class of utterances, because some of the cases may or may not be present, cases can appear in any order, each case can have several different possible fillers, and each semantic relation can have several different possible grammatical representations.

In our system, case frames are never explicitly written by a user. Instead, we use the concept hierarchy of YAKS to build a corresponding case frame hierarchy (there are also case frames for the roles), along which cases are implicitly inherited. A similar technique is used for the fillers: Instead of individual words, concepts are stated as allowed fillers. With any concept all of its superconcepts and subconcepts are legal fillers, too. For each of these concepts, a whole set of words or phrases can be given as natural language representations. Case frames can be nested when parsing: If, for example, the filler of a case in a case frame is a noun that has a case frame associated with it, a complete instantiation of this case frame can be filled into that case. The grammatical representations that are possible for each of the semantic relations are listed in a separate *case table*. Entries in this case table have separate parts for occurrences of the case in nominal and in verbal context, if applicable, which enables to parse either a complete sentence or the corresponding nominalization with the same case frame. This representation avoids tedious repetition and makes the representation compact and almost free of redundancy.

5 SARA Knowledge Acquisition

One reason why most natural language interfaces are not successful is that it require too much work to adapt them to a new domain. In the design of YAKR, we thus paid special attention to the problem of knowledge acquisition for the natural language interface: Ideally, nothing more should be necessary to acquire than the words which can be used to refer to each concept, role or instance. If this ideal is not approximated closely enough, it is necessary to specify complex grammatical descriptions; A4 tells us that in this case the natural language interface will not be successful in practice.

But YAKR is very close to the ideal: The knowledge acquisition for the natural language interface in our system consists of adding short annotations to each concept definition or role definition in the knowledge base. There are three main types of information present in the annotations: (1) information about individual words, (2) information about cases, and (3) information about explicit inheritance of words.

The word information associates each concept or role with its natural language synonyms and with simple phrases that represent this concept or role. The variety of the phrases covered by the annotations to a concept or role X is increased by using an inheritance mechanism to derive parts of these phrases from the annotations of superconcepts or explicitly stated syntactical superroles of X.

The information about cases enables the generation of the case frames themselves. It describes which cases must be inserted into which case frames. This annotation is needed for roles only. Nothing more than the name of a case table entry has to be given, since the set of case frames to put the case into and the corresponding allowed filler for each insertion can be deduced from the YAKS modeling.

The information about explicit inheritance of words is used for roles and for derived concepts in order to be able to inherit nouns for the description of multiword phrases for these concepts or roles, although there is no hierarchical predecessor (i.e. superrole or superconcept) from which the nouns could be taken.

The annotations are now described one by one; please refer to the following example. It should be noticed that this example was not made for a real application but to show how YAKR works. It does thus not show a stringent modeling style. A detailed (although not completely up-to-date) description of the annotations and their semantics can be found in [9].

```
PRIM-CONCEPT Objects = anything.
   NOUN(komponente objekt).
   ROLES(has-synonym : strings;
         NOUN(name bezeichner identifizierer) ).

PRIM-CONCEPT actions = anything.
   NOUN(aktion ereignis vorgang).
   ROLES(has-agent : Functions;
         AT-DOMAIN(agent) ).

PRIM-CONCEPT Call-Actions = actions.
   VERB(rufen (rufen auf))
   NOUN(aufruf aufrufen ruf rufen)
   PREFIX(aufruf).
   ROLES(calls : Functions;
         AT-DOMAIN(thema)
         with : Parameters;
```

```
         AT-DOMAIN(attribut) ).

PRIM-CONCEPT Functions = Objects.
   NOUN(unterprogramm funktion prozedur).
   ROLES(has-Parameter : Data-Objects;
         NOUN(parameter) ).

DEF-CONCEPT Input-Functions = AND(Functions
                                  SOME(reads)).
   PREFIX(eingabe einlesen lesen input)
   SYN-CASE((zweck reads))
   ADJECTIVE(einlesend lesend).
   ROLES(reads : Data-Objects;
         VERB(lesen (lesen ein))
         NOUN(lesen eingabe einlesen input) ).

DEF-CONCEPT Output-Functions = AND(Functions
                                   SOME(writes)).
   PREFIX(ausgabe ausgeben schreiben output)
   SYN-CASE((zweck writes))
   ADJECTIVE(ausgebend).
   ROLES(writes : Data-Objects;
         VERB(schreiben (geben aus))
         NOUN(schreiben ausgabe ausgeben output) ).

PRIM-CONCEPT Data-Objects = Objects.
   NOUN(daten)
   PREFIX(daten).

DEF-CONCEPT Parameters = AND(Data-Objects
                             SOME(is-Parameter-of)).
   NOUN(argument parameter)
   ADJECTIVE(uebergeben).
   ROLES(is-Parameter-of = INV(has-Parameter); ).

PRIM-CONCEPT Characters = Data-Objects.
   NOUN(zeichen char character).

PRIM-CONCEPT Lines = Data-Objects.
   NOUN(zeile).

PRIM-CONCEPT Files = Data-Objects.
   NOUN(datei file).

INSTANCE character-c = Characters.
INSTANCE string-s = Lines.
INSTANCE filepointer-stream = Files.

INSTANCE fgetc = AND(Functions
```

```
                    VALUES(reads, [character-c])
                    VALUES(has-Parameter, [filepointer-stream])
                    VALUES(has-synonym, ["fgetc"])).

INSTANCE fputc = AND(Functions
                    VALUES(writes, [character-c])
                    VALUES(has-Parameter, [character-c filepointer-stream])
                    VALUES(has-synonym, ["fputc"])).

INSTANCE fgets = AND(Functions
                    VALUES(reads, [string-s])
                    VALUES(has-Parameter, [string-s filepointer-stream])
                    VALUES(has-synonym, ["fgets"])).

INSTANCE fputs = AND(Functions
                    VALUES(writes, [string-s])
                    VALUES(has-Parameter, [string-s filepointer-stream])
                    VALUES(has-synonym, ["fputs"])).
```

Note that the concepts *actions* and *Call-Actions* and their accompanying roles are not really used in this example; they are present for explanation purposes only.

The hierarchy that results from this example is depicted in figure 2. The roles *calls* and *with* are not shown in this picture, because they are not actually used in the example anyway.

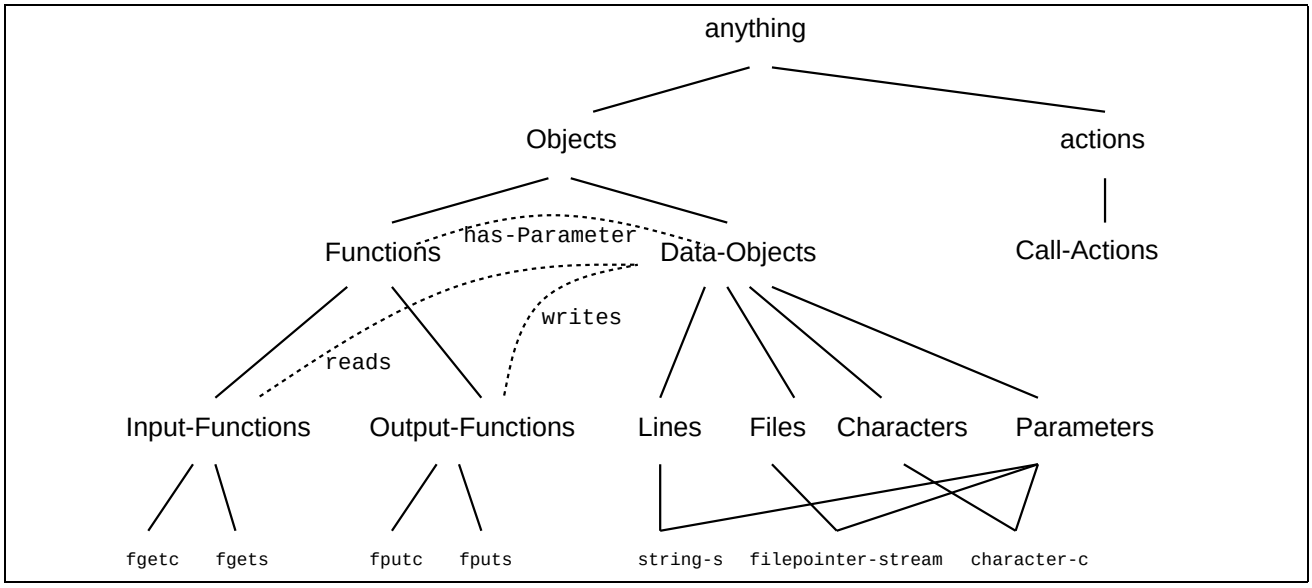

Figure 2: Taxonomy of the example knowledge base

5.1 Information about Individual Words

The simplest form of annotation to a concept or role X is the synonym list: The `VERB` and `NOUN` annotations give a list of verbs and non-compound nouns, respectively, that can denote the concept or role X they annotate. The same word can annotate multiple concepts or roles, resulting in ambiguities

for that word. For instance, the german words Funktion (function) and Unterprogramm (subprogram) both refer to the concept *Functions*. Similarly, lesen (to read) and einlesen (to read in) refer to the role *reads*. einlesen is a verb with a separable prefix and is therefore given in two parts.

Often nouns can be specialized by prefixing an adjective. The `ADJECTIVE` annotation to X expresses this prefixing: It lists a number of adjectives that can be used to specialize a noun in order to denote X. The suitable nouns for this specialization are all nouns that annotate any superconcept of X. The `ADJECTIVE` annotation shows one of the ways inheritance is used in the construction of case frames: Whenever an annotation to X specifies a part P of a natural language construct to be *added* to another construct A, then A is inherited from the superconcepts or superroles of X. For instance, lesende Funktion (reading function) refers to *Input-Functions*, where Funktion is inherited from the annotation of the direct superconcept *Functions*. The word lesend is a present participle, but can be used as an adjective in our system. The nouns that can be used need not be annotated at direct superconcepts: lesendes Objekt (reading object) could be used as well to denote *Input-Functions*. In a real knowledge base this phrase would most probably be ambiguous, because lesend might annotate other subconcepts of *Objects* as well, but ambiguity a good result in this case, because the phrase is indeed very vague.

Compound nouns are written as a single word in German, so they could all be put into the dictionary and just annotated in the `NOUN` list. However, this would be extremely tedious, since compound nouns are very versatile and ubiquitous in German. To solve this problem, the `PREFIX` annotation to X lists possible prefix nouns that can be prepended to the nouns annotated with the superconcepts of X in order to form a reference to X. This annotation is analogous to the `ADJECTIVE` annotation. For instance, Lesefunktion (read function) refers to *Input-Function*. Only the words Lesen[2] and Funktion need to be in the dictionary, the compound is algorithmically broken into these components by the parser. As with adjectives, inheritance is possible from concepts that are more than one level above.

Instead of using an adjective or noun prefix to specialize a noun, it is often possible to use a prepositional phrase that is placed right behind the noun. This possibility can be expressed with the `SYN-CASE` (synonym case) annotation to X: It gives a case (with a filler) that, when used together with a noun that is annotated to a superconcept of X, denotes X. This annotation is an extension of the usual case frame formalism, because it produces a case that does not get inserted into a case frame instantiation, but modifies a filler or head instead. For instance Funktion zum Lesen (function for reading) refers to *Input-Functions*. The possible grammatical forms for this reference are listed in the case table entry *zweck* (see appendix A.2); its relevant entry for this phrase is *"preposition* zum *with dative case"*. This type of annotation need not be used, because the same handling capability can be achieved by integrating an additional role into the model: zweck means purpose and could have been modeled with a role *has-purpose* that has *Functions* in its domain. But a model may be somewhat superficial in some parts; then the `SYN-CASE` annotation is a simple way to increase the coverage of the natural language interface.

5.2 Information about Cases

All the above annotations merely describe phrases that represent pure concepts or roles; no case frames are built from them. The source of cases for the case frames are the roles. Information about cases is annotated for roles only and tells what cases to insert where.

There are two different case annotations: `AT-DOMAIN(sc)` at a role R creates a case in the case frame of the domain concept of R. The allowed filler for this case is the range concept of R; it has to appear

[2] In fact, it is even sufficient to have the *verb* lesen in the dictionary, because its nominalization is automatically inferred. See appendix A.1

in a grammatical form described by the role table entry *sc* (see appendix A.2). The dual form of this annotation is `AT-RANGE(sc)`. Given at a role R, the `AT-RANGE` annotation creates a case in the case frame of the range concept of R. The allowed filler for this case is the domain concept of R; it has to appear in a grammatical form described by the role table entry *sc*. Examples can be found in section 6.1. Note that the cases are inherited by subconcepts of the concepts they originally target at.

Not all cases in all case frames are created from such annotations. Some cases can be added without annotations and some case frames can be built completely automatically. These details are explained in section 6.1 below.

5.3 Information about Explicit Inheritance of Words

Since roles are not arranged into a hierarchy in YAKS, there are no superroles for a role and it is not directly possible to use the `PREFIX` and `ADJECTIVE` annotations, because they rely on inheritance. A similar statement is true for derived concepts, because, since a derived concept D is described by conditions that are sufficient but not necessary, no concept can ever be guaranteed to be a superconcept of D (i.e. to include all instances that D includes). To overcome this problem, there is the `SUPERR` annotation to roles (and the analogous `SUPERC` annotation to concepts). To annotate `SUPERR(S)` at a role R means, that the role R shall inherit nouns from S. Although this form of annotation may seem a bit unsatisfactory, it is still much better than explicit annotation of complete phrases, because it is much more compact and retains as much of the usual way the annotations work as possible. `SUPERR` and `SUPERC` annotations are not inherited, i.e. they cannot be used across multiple levels.

6 Integration of YAKS and SARA

As mentioned above, the knowledge acquisition for the natural language interface in our system consists of adding annotations to the knowledge base. In this section we describe how these annotations are used to construct the case frames and how instantiated case frames are converted into a YAKS query after parsing.

There is a separate system component in YAKR for each of these tasks: The case frame generator builds the case frames for SARA using the annotations and the YAKS knowledge base itself. The case frame interpreter transforms instantiated case frames as produced by the parser into queries that are processable by YAKS. In our current YAKR prototype, answers are the results of such queries.

6.1 Generation of Case Frames

This subsection describes, how the YAKS knowledge base and the annotations are used to generate all the case frames that the parser SARA uses and what the generated case frames look like.

Not all concepts and roles can be handled the same way. For instance, concepts describing a property require case frames with sein (to be) and haben (to have) as head, while other concepts do not. Thus the basic criterium for which case frames to generate is the *category* of a concept or role; the following categories are distinguished:
The *string concepts* and *integer concepts* are subconcepts of *string* or *integer*, respectively; no case frames are generated for them. *Attribute concepts* are subconcepts of *specifiers*. All other concepts are either action or object concepts and are handled the same; they belong to the category *action-or-object concepts*.
The *has-attribute roles* have *specifiers* as their range. The *synonymless roles* have *actions* as their

domain. The *theme roles* have verb synonyms. All these conditions are necessary and sufficient (otherwise the modeling is incorrect). The *has-action-or-object roles* have *actions* or *objects* as their range.

For each of these categories there are fixed rules that describe which cases and case frames must be generated; the case frame generator module implements these rules. A case frame consists of a head and a set of cases. Here are two examples:

```
Call-Actions  [C-Call-Actions]  (
  agent has-agent (C-Functions),
  thema calls (C-Functions),
  attribut with (C-Parameters),
  benennung has-synonym (C-Call-Actions))

has-Parameter_HAOd  [R-has-Parameter]  (
  agent DR (C-Functions) (21),
  thema RR (R-has-Parameter) (21))
```

`Call-Actions` and `has-Parameter_HAOd` are the name of the first and second case frame, respectively. `Call-Actions/has-Parameter` is the head of the first/second case frame (marked to be a concept/role). Each of the indented lines is one case. The components of a case are, in the order shown, the syntactical role (i.e. the name of an entry in SARA's role table; see appendix A.2), the YAKS expression to be used to generate that part of the YAKS query from the instantiated case frame that corresponds to this case (if it is filled), the list of allowed fillers for this case (which most often contains only one element), and optionally the priority mark, which is 20 (and not shown then) by default; a value of 21 marks a case as mandatory, i.e. it must appear in an instantiation or else that instantiation is illegal. See appendix B for a complete listing of the case frames that are generated from the example modeling.

We now describe for each of the relevant concept and role categories what cases and case frames are generated by the case frame generator:

6.1.1 Action-or-Object Concepts

One case frame is generated for every action-or-object concept *AO*. It contains at least one case of the form

`benennung has-synonym (C-`*AO*`)`

This case is meant to catch names behind a noun in phrases such as die Funktion "f", where it catches the "f". Other cases may be present due to `AT-DOMAIN/AT-RANGE` annotations: All roles *R* with an `AT-DOMAIN(sc)` annotation that have an action-or-object concept as their domain and a concept *rng* as their range generate a case of the form

`sc` *R* `(C-`*rng*`)`

and analogously all roles *R* with an `AT-RANGE(sc)` annotation that have an action-or-object concept as their range and a concept *dom* as their domain generate a case of the form

`sc INV(`*R*`) (C-`*dom*`)`

In the above example, `Call-Actions` is such a case frame.

6.1.2 Attribute Concepts

Attribute concepts are handled the same way as action-or-object concepts, but usually no roles with an `AT-DOMAIN` annotation have an attribute concept as their domain and no roles with an `AT-RANGE`

annotation have an attribute concept as their range. Thus there are usually no cases in the case frame of an attribute concept except the one to catch a name.

6.1.3 Has-Action-or-Object Roles

For each has-action-or-object role three case frames are generated. These are named after the role with additional suffixes *_HAOa*, *_HAOc*, and *_HAOd*[3]. As an example, assume the domain *Functions* of the role *has-Parameter* has been annotated with the noun Funktion, the range *Data-Objects* with Datenobjekt, and the role itself with Parameter, then

- the *has-Parameter_HAOa* case frame serves to parse nominal phrases that mention the role itself as a relation such as der Parameter "X" von Funktion "f",
- *has-Parameter_HAOc* parses to-be phrases of the kind der Parameter ist das Datenobjekt "B" (using *has-Parameter_HAOa* to catch the Parameter), and
- *has-Parameter_HAOd* parses to-have phrases of the kind die Funktion "f" hat den Parameter "P" (using *has-Parameter_HAOa* to catch the Parameter "P"), which is the most natural paraphrase of the relation described by the role itself.

Of course the actual inputs will usually not be declarative sentences. The *R_HAOa* case frame of a role *R* with domain *dom* and range *rng* uses the role itself as its head and has the two cases

`benennung has-synonym (C-`*rng*`)`
`gen_von INV(`*R*`) (C-`*dom*`)`

The *R_HAOc* case frame of a role *R* with range *rng* uses the word sein as its head at parsing time and has the two cases

`agent RR (R-`*R*`) (21)`
`definition RR (C-`*rng*`) (21)`

Where the allowed filler of the *agent* case means that the *R_HAOa* case frame of the same role must be used and the syntactic role *definition* stands for the grammatical case "nominative". Both cases are mandatory (i.e. must be filled for an instantiation to be legal). The *R_HAOd* case frame of a role *R* with domain *dom* uses the word haben as its head at parsing time and has the two cases

`agent DR (C-`*dom*`)`
`thema RR (R-`*R*`)`

Where the allowed filler of the *thema* case means that the *R_HAOa* case frame of the same role must be used and the syntactic role *thema* stands for the grammatical case "accusative". `DR` stands for "domain restriction" and `RR` for "range restriction"; the instantiated case frame is transformed into a role expression by the query generator.

6.1.4 Has-Attribute Roles

has-attribute roles are handled much as has-action-or-object roles with the following differences: (a) the suffixes of the case frame names are *_HAa*, *_HAc*, and *_HAd* and (b) since concrete specifiers usually appear as adjectives, the *definition* case in the *_HAOc* case frame is not always sufficient to catch the range of the role; it is therefore complemented by another case with the syntactic role *adj_adv* (adjective or adverb) and exactly one of these two cases must be filled by an input sentence.

[3]Once upon a time, a *_HAOb* case frame existed, too, but it has been merged into the *_HAOa* case frame now.

6.1.5 Theme Roles

Theme roles generate one simple case frame that contains exactly two cases: one for the domain D of the role and one for its range R. These cases have always the same syntactical roles and YAKS expressions associated with them; they look like the following:

`agent` `DR` `(C-`D`)`
`thema` `RR` `(C-`R`)`

Again, `DR` stands for "domain restriction" and `RR` for "range restriction".

6.1.6 Synonymless Roles

As their name says, synonymless roles have no word annotations. Thus it is not possible for a case frame of such a role to be activated by a certain word in an input sentence. Consequently there are no case frames for synonymless roles, but a synonymless role always has an `AT-DOMAIN` annotation to insert a case into the case frame of an action concept.

6.2 Generation of Queries

This subsection describes how the instantitated case frames generated from an input sentence by the parser are translated into YAKS queries. A detailed description of this translation can be found in [9].

Generally speaking, the head of a case frame instantiation, as well as all of its fillers are considered restrictions. These restrictions are computed and combined into a query expression — in many cases simply by applying the `AND` operator to them. The computation of the restrictions is recursive; the recursion ends at individual words (to be precise: at those elements of a case frame instantiation that stem from individual words) that denote a concept, role, or instance.

There are two possible forms of queries: concept queries and role queries. First of all it must be determined which of these types should be generated from any given case frame. Role queries can provide more information, since they show associations of objects in the answer. We thus try to generate role queries as often as possible, i.e., whenever we have an idea about which pairs might be interesting. Role queries are generated from (a) all case frame instantiations that have sein or haben as head words, (b) all wh-questions, and (c) pure simple noun phrases that are annotated at a role, i.e. instantiations of *_HAOa* and *_HAa* case frames that have no case filled.

The basic translations in the recursive process of building a query are the following:

1. Head concepts of instantiations translate into themselves.
2. Head roles of instantiations translate into themselves, except for instantiations of *_HAOa* and *_HAa* case frames, where they stand for the range of the role whenever any cases are filled or the level of recursion is deeper than 1. So if the input consisted of nothing more than a synonym of a *has*-role, the query will be a role query, instead of a query for the range of that role, because role queries yield more information.
3. A case with syntactical role *benennung* and YAKS expression *has-synonym*, i.e. a case filled by a name X, is translated into
   `VALUES(has-synonym, [''`X`''])`
4. A case with YAKS expression DR and filler X is translated into
   `DR(`X`)`
   RR is handled analogously.

5. Any other case with YAKS expression *expr* and filler X is translated into
   `SOME(expr,` X`)`

Somewhat different handling is necessary for relative clauses and for the construction of appropriate role queries for Wh-questions. This handling is sketched in the following paragraphs.

The restriction that is defined by a relative clause has to be put into the restriction that is returned for the noun to which the relative pronoun refers. This is straightforward if the relative clause maps into a concept expression. But if it would normally map into a role expression (because the head of the instantiation for the relative clause is a role R), it has to be converted into a equivalent concept expression. It is possible to do so, because we know whether the relative pronoun filled (a) a case marked with DR or (b) one marked with RR: Let the translation of the filler of the other case in the relative clause be F, then return

`SOME(`R`,` F`)` for (a), or
`SOME(INV(`R`),` F`)` for (b).

If F should be empty, we use the most general concept *anything* instead.

For Wh-questions it is necessary to defer the generation of the query terms that correspond to the case that is asked for until the rest of the query is known. We therefore propagate markers for the asked role R and its filler F towards the uppermost level of the recursive process. There we can then produce an appropriate role query from R, F, and the rest Q of the query restrictions as

`ANDR(DR(`Q`)` R `RR(`F`))`

Actually, we generate the slightly more efficient equivalent specialized form

`DRR(`Q`,` R`,` F`)`

In a similar way, it is also possible to build correct queries from explicit questions for pairs, between which no role actually exists in the modeling by composing the two roles that the question asks for.

7 Experiences and Limitations

7.1 Usefulness

With a well modeled, well annotated, sufficiently big database the usefulness of YAKR for an end-user is high: YAKR allows to search a repository with a short query that is easy to formulate without mastering a complex special query language. The recall of such a query is usually high, since a taxonomy and ambiguous vocabulary can be used. If the precision is not high enough in the first attempt, it can be raised by refining the natural language query, (e.g. adding or specializing an attribute), which is easy for a user even in complicated cases.

The usefulness of the current implementation, however, is reduced, because the syntactic and semantic coverage of our system is quite limited. As pointed out in [5], a limited natural language interface may still be useful as long as there is at least one way of expressing the thing wanted and the user can easily find that term or phrase. It is important that the user can easily learn the restrictions of the interface and learn how to avoid unprocessable input. How well these conditions are met by our system is impossible to judge now, since we have not gathered enough experience yet.

To improve the usefulness of YAKR, SARA could be replaced by a more sophisticated natural language processor. It is not clear how easily the annotations could be extended to cover more complex natural language utterances. Many extensions will most probably be straightforward: For example, the restricted natural language that our system processes is currently more limited than the query language provided by YAKS, since conjunctions cannot be handled; a corresponding extension would be useful. It would be no problem to replace our natural language processing component with one that is more

sophisticated in this respect, without any need to change the annotations at all. On the other hand, it will, for example, be difficult to describe classes of paraphrases by annotations.

7.2 Practicability

The practicability of the implementation of a system such as YAKR is good: The software is of moderate size and could be produced by a small team in some months. It took about 4 person years (including the design); a re-implementation could probably done in half the time. Even our prototype is neither too buggy nor too slow to use it.

The practicability of the database maintenance is difficult to predict in general. If additions to the database are homogeneous in the sense that they do not require changes in the terminology, they are no problem; maintenance in this case is about as difficult as for any other database with a nontrivial schema. If, on the other hand, components will be added that require new terminology, some expertise in modeling is needed to keep the terminology clean and coherent. Another problem is the masterability of a very large terminology, which is needed if the database contains adequate descriptions of software components of many very different kinds.

7.3 Acceptance

We expect that in principle the acceptance of a system such as YAKR would be high as far as the usefulness for the end user is concerned, but may be low as far as the practicability of database maintenance is concerned (see the two previous sections). It may, however, make sense, to use the system in an encapsulated form, i.e., with a more or less fixed knowledge base.

End-user acceptance of our prototype might be limited by the restrictions of its syntactic and semantic coverage as discussed above.

7.4 Scalability

The scalability of the approach is good, if it is not too difficult to master a big terminology; we do not yet know whether this is the case. The method of retrieval allows to keep both precision and recall high even for large databases, because the data is accessed semantically and can be differentiated with arbitrary granularity. Our modeling experience indicates that taxonomies tend to grow more in width than in height. Since the run time for a deduction depends most on the number of instances that belong to the relevant branches of the taxonomy, but only rarely on the overall size of the knowledge base, response times would not necessarily get too long, either.

The scalability of our current implementation, however, is problematic, because we currently use a lot of memory for any single concept, role, or instance. The subset of the NIH-Class-Library we have modeled uses about 200 concepts, 130 roles and has about 1000 instances; response times are between below one second and about 30 seconds on a SUN-4 workstation. The YAKR process needs about 13 Megabytes of memory after the knowledge base has been loaded. Using more instances with the same concepts and roles would increase memory needs linearly and response time slightly sublinearly.

7.5 Applicability

The approach used in YAKR is not applicable to all kinds of software reuse equally well. The model of reuse we had in mind when we designed YAKR, was what is usually called “late reuse”: When components were built, no special arrangements were made to reuse them, except perhaps providing

some human-readable documentation for them. The repository we target consists of components for which no formal input/output specification is available and which do not necessarily use common data structures, common processing models, or common modularization strategies. Thus the informality in the information that is available about the components shows up in our system in the informality of the querying interface we use.

On the other hand, reuse could also take place in a more controlled and formalized environment, where the production (and catalogization) of reusable components and the production of new software using them are held together by a common formal framework. In this case other methods to access a repository of these components might be superior, namely those that use the information that is implicitly represented in that framework. Even in this case, however, YAKS may be a good tool to represent the repository, only the access method should be extended.

Note that the applicability of YAKR extends beyond what is usually understood as software reuse: in cooperation with `CPPREF` it is a valuable aid in the process of changing a software system, too. Changes, whether major or minor, in existing software systems could of course be considered software reuse.

8 Related Work

8.1 Finding Reusable Components

The main concern of the Software Information System is to find reusable components. There is of course a lot of other work on this problem, often with completely different approaches. This section surveys some of the most important of them and compares them to the approach taken in YAKR.

The probably most well-known work in representing and finding reusable software components is that of Prieto-Diaz [26, 25, 27, 28]. He proposes a so-called "faceted classification" of software components: To classify a component, from any one of a small set of predefined facets (aspects), exactly one of a set of predefined terms must be chosen. Six facets are proposed: Three concerning the functionality of the component, namely *function, object*, and *medium* and three concerning its environment, namely *system type, functional area*, and *setting*. A query is thus a sixtuple of terms from these facets. The some dozens of terms within each facet are arranged into a so called *conceptual graph*, which describes the distance between each supertype–subtype pair of terms and is used to define *conceptual closeness*. This closeness measure is used to find components that can be regarded as similar if no exact match for the query tuple can be found. To avoid problems stemming from a fixed set of terms for each facet, i.e. a controlled vocabulary, a thesaurus is used. The resulting set of candidate components is then ranked by an evalutation mechanism, which uses additional information about the component (size, structure, quality of documentation, implementation language) and a user profile (describing the user's reuse experience) to identify those components with the lowest reuse effort.

Compared to YAKR, Prieto-Diaz' system is more complete: YAKR neither adapts to user experience nor in any way ranks or evaluates the answers it finds. Knowledge acquisition effort will often be higher in YAKR, depending on the granularity of modeling. On the other hand, YAKR has the advantage that it does not assume a user's familiarity with the internals of the model; no such thing as a faceted classification scheme has to be understood by a user. The deductive capabilities make the retrieval in YAKR more powerful, too, because the concept hierarchy of YAKR contains much more information than the conceptual closeness graph, since it also considers the relations expressed by roles, and it does not have to be computed "by hand". A natural language query is also quicker to produce than a sixtuple can be built.

The software information system that is most similar to ours is LaSSIE/CODE-BASE [14, 32]. LaSSIE originally used a frame based knowledge representation language called KANDOR, which was later replaced by a language called Classic. In LaSSIE all information in the knowledge base had to be handcoded. In LaSSIE's successor, CODE-BASE, a lot of information is acquired by an automatic tool, similar to CPPREF, and is then stored in a database which is queried on demand. The user interface of CODE-BASE consists of a natural language parser plus a graphical browser for navigating in the concept hierarchy.

While KANDOR is similar to YAKS in its design, some powerful constructs are missing; for example the transitive closure of a role, the composition of two roles, the inversion of roles, union of concepts or roles, and querying with role expressions. Thus KANDOR's expressive power is considerably smaller than that of YAKS. These deficiencies have been reduced with the use of Classic; but some gaps remain even in the newest version[4]: Composition of roles is allowed only for roles with at most one filler in Classic. Concept or role union, transitive closure of roles, negation of concepts or roles, and intersection of roles are not supported.

While CODE-BASE has a useful graphical interface that YAKR has not and avoids storing the whole knowledge base in main memory, the main advantage of YAKR compared to LaSSIE/CODE-BASE is the transportability design of the natural language interface. For a short discussion of the LaSSIE natural language interface, see the next section.

A simpler variant of this approach is represented by the work of Wook and Sommerville [39]. They use conceptual dependency theory [31] (a canonicalized variant of case frames) to represent the components. After the user has given a keyword for the main action of the needed component, the system presents a template of the corresponding conceptual dependency frame, which is then interactively filled by the user and used for component retrieval. Each component is assigned to one of 5 categories in the internal modeling. Each of these categories has a fixed set of slots for its corresponding conceptual dependency frame.

This system lacks the inference capabilities of YAKS and the capability to express subtle semantic differences. While the user interface has the advantage to guarantee that the input is meaningful and understandable for the system, it is badly suited for complex components with a large number of slots.

There are lots of other approaches to the problem of finding reusable components. Most of them are more or less lacking the capability to handle requests of a user, who has only an incomplete and fuzzy specification of the needed component at hand. Many of them are not general: All approaches that rely on formal specifications in any way, e.g. [30], are completely impractical for interactive components, e.g. X Window System widgets, and are difficult to use. Others, e.g. [23], require the user to specify the component "by example", which seems useful as an additional technique (if the user knows the functionality needed, but has no vocabulary to describe it), but most often places far too much workload on the user and is also impractical for components that cannot easily be described by their input/output behavior.

8.2 Transportable Natural Language Interfaces

An important result of the SIS project is the annotation approach to the construction of transportable natural language interfaces. This section compares related approaches to this problem.

TEAM [20] is probably the most sophisticated and well known transportable natural language interface. TEAM makes use of four knowledge sources: the augmented phrase structure grammar, the lexicon, the conceptual schema, and the database schema. The grammar is fixed and has a broad

[4] Peter Patel-Schneider, personal communication, June 1992

syntactic coverage; the other knowledge sources have to be updated for a new database. The lexicon contains word information for morphological, syntactic, and semantic processing. The conceptual schema consists of sort information and constraints on the arguments of nonsort predicates. Finally, the database schema consists of information that enables the mapping of the intermediate representation to a query expressed in a relational query language.

The acquisition process differs from the one we use. Our approach relies on specifying lexical and morphological knowledge in the lexicon and annotated knowledge structures. In TEAM, lexical and syntactic knowledge is acquired by means of an acquisition dialogue using menus and windows. The dialogue is guided by the knowledge structures and the answers of the user. Verbal case frames are acquired with an example sentence given by the knowledge engineer and questions about correctness of sentences generated by the system. This kind of acquisition is motivated by the aim that non-natural-language-experts should be able to adapt the interface to a new database. Similar acquisition tools are not available in our system and updates are currently more difficult. However, building such tools for our system would be no problem.

YAKS permits a terser model than the relational language of TEAM does. Furthermore, information that has to be explicitly acquired for the interface in TEAM may be deduced when using YAKS. For example, we do not need an extra sort hierarchy and can represent arithmetic fields just as relations that have the concept 'integers' as their range. Hence, the knowledge acquisition process requires less effort than it does in TEAM.

However, our system has some limitations compared to TEAM. Our parser has a more restricted syntactic coverage and does not process quantifiers, numbers, connectives, comparatives, and anaphorical references. We also do not handle the problems that these syntactic constructions entail in semantic processing, such as scope determination. Furthermore, our system does not support measurement units and does not have special mechanisms for handling relations or cases related to time or location.

Chat-80 [37] is an efficient, transportable system for Prolog knowledge bases. The system is implemented in Prolog itself. The lexicon and the case information are specified with Prolog clauses. This specification is tedious compared to our mechanism. As in TEAM, extra type information must be specified. The syntactic component is, like ours, rather restricted and does not cover anaphorical references and most conjunctions.

Another integration of a natural language interface with a KL-ONE-like knowledge base has been done for the LaSSIE system and is described rather briefly in [15]. The natural language component is realized by customizing the natural language processor TELI [4]. The knowledge structures of TELI are similar to those used in SARA, but there is no inheritance hierarchy for the case frames. The integration does not work with general models and is restricted to frames describing actions. Most important of all, the transportation of this interface to a new domain requires much more work than in YAKR.

9 Results

Our results can be summarized as follows:

1. It is possible to define a knowledge representation language that is powerful enough to describe software components in a useful form.
2. Modeling software in such a language is not easy.
3. Such a knowledge representation language can be implemented with reasonable effort and shows acceptable efficiency.

4. The practical efficiency of the deductions varies much with a large knowledge base: Many queries return within less than a second, some others take minutes.

5. It is possible to build a natural language interface for a specific knowledge base with only minimal additional work (less than 10 percent) for the knowledge engineer.

6. A natural language interface to a repository of software components is useful to have, even if it is syntactically restricted.

10 Further Work

There are obviously a lot of possibilities to improve our system. The most important ones would be to augment the capabilities of the natural language interface (syntactic and semantic), to complement the natural language interface with menu and windowing techniques (for instance to access the source code or documentation files), to speed up those deductions that are now very costly, and to avoid putting the whole knowledge base into virtual memory. We do not currently plan to follow any of these paths.

But there is another issue of striking importance: Is this semantic modeling plus natural language approach to find reusable components really better than, for instance, much cheaper information retrieval methods that rely on fulltext keyword search on documentation files of the components ?

We will perform an experiment to investigate this issue: A part of the programs described in chapter 1 of the UNIX user manual will be modeled. The precision and recall of our system will then be compared to keyword search on the full text of the manual pages of these programs. We expect that the result of this experiment will depend heavily on the definitions of what "equivalent queries" and "unsuccessful query" means, because a natural language interface opens a lot of opportunities for "near misses". Thus the result will depend much on interpretation, but should nevertheless give us an idea where our knowledge based approach is strong and where (and why) it is not.

Acknowledgements

Rolf Adams initiated the SIS project and performed most of the conceptual work; he also implemented YAKS. Martin Fischer implemented the KRS prototype. Lutz Prechelt designed and implemented SARA and implemented the annotations and the case frame generator. Finn Dag Buø together with Rolf Adams and Lutz Prechelt designed the annotations, case frame generator, and case frame interpreter. He also implemented the latter. Walter Tichy supervised the project. Stuart Feldman, GNU, Heinz Herrmann, Brian Kernighan, Donald Knuth, Dennis Ritchie, Ken Thompson, Richard Stallman, Bjarne Stroustrup, SUN Microsystems, Walter Tichy, Universität Karlsruhe, and lots of others supplied a sufficiently nice working environment.

A part of the SIS project was sponsored by Siemens AG, ZFE BT SE 3.

A Other knowledge sources of YAKR

Apart from the YAKS specification of the knowledge base, there are two other sources of knowledge to the system, both needed by the parser: the dictionary of word forms and the table of syntactic role forms. Their formats and semantics are described in the following two subsections.

A.1 SARA dictionary

The dictionary contains all the words that SARA shall be able to recognize. It is implemented partly as a full form dictionary and partly with algorithmic word form analysis. The information it delivers to the parser are (where appropriate): part-of-speech label, time, casus, numerus, genus, person, comparation, and contained prefix. For most types of words, dictionary information rarely needs to be changed (see the file `saralib/sara.std`). But for verbs, nouns, and adjectives additions are necessary frequently, although the dictionary already contains about 10000 words with about 25000 explicitly stored word forms plus several times as many implicitly recognizable word forms. The format of entries for these are thus described here. All explanations are by example for ease of understanding.

A.1.1 Verbs

Irregular verbs are all defined in the file `saralib/sara.uverben`, their format can be deduced from example there and the format for regular verbs as described below.

Entries for regular verbs are collected in the file `saralib/sis.wb.v` and may look like

```
{ ackern   :verb  :praefixe (be)  :halbpraefixe ()          :ung }
{ addieren :verb  :praefixe (-)   :halbpraefixe (auf)       :rm }
{ agieren  :verb  :praefixe (- inter re)  :halbpraefixe () :rm }
{ fertigen :verb  :praefixe (ver) :halbpraefixe (an ab aus vor) :ung }
```

The first part of the entry is the *name* of the dictionary entry and must be the infinitive form of the verb.

`:verb` is the key word that assigns the part-of-speech.

`:praefixe` with a following parenthesized list of blank- or comma-separated prefix word parts is optional and designates other verbs with the same stem part (in the example ackern and beackern). Note that the Partizip II form is different for prefixed verbs: ackern → geackert but beackern → beackert (and not gebeackert). The dash designates the empty prefix that has the meaning "do not prepend ge in the Partizip II form", e.g. addiert instead of geaddiert. It can be given even where other prefixes are also needed and is usually necessary for verbs with foreign origin.

`:halbpraefixe` with a following parenthesized blank- or comma-separated list of separable prefixes is optional, too, and designates even more other verbs with the same stem part (in the example fertigen and abfertigen). The difference between separable prefix and full prefix is that a separable prefix is cut off from the rest of the verb form in present tense, e.g., ich fertige etwas ab but not ich abfertige etwas Note that the Partizip II form is again different for verbs with separable prefix (except if the empty full prefix is present) because the ge is inserted *between* the prefix and the stem: fertigen → gefertigt but anfertigen → angefertigt (and not geanfertigt). The dash in the `:praefix` list eliminates the ge also in the Partizip II forms with separable prefix, e.g., aufaddiert instead of aufgeaddiert.
Only the Partizip I and Partizp II forms of verbs with separable prefix are explicitly generated in the dictionary as full forms; all others are recognized via algorithmic word form analysis.

The last part of a verb entry is either :rm (which is an abbreviation for the also possible :regelmaessig) or :ung. :rm designates the verb as a regular one. :ung does the same, but additionally results in the generation of another nominalization: For all verbs (whether regular or not), the infinitive form is automatically put into the dictionary as a noun, too (e.g. ackern → das Ackern). For :ung verbs second noun entry is made, in which the en ending of the infinitive form is replaced by ung, e.g., fertigen → das Fertigen,die Fertigung.

The Partizip I and Partizip II forms of all verbs are automatically also available as adjectives and as adverbs.

A.1.2 Nouns

Entries for nouns may look like

```
{ abschnitt :substantiv :typ (Ss, Pe) }
{ adresse   :sub    :typ (S, Pn) }
{ bild      :sub    :geschlecht (s)  :typ (Ss, Per) }
{ algorithmus :sub  :geschlecht (m)  :typ (S) }
{ algorithmus :sub  :stamm algorithmen   :geschlecht (m)  :typ (P) }
{ solo      :sub    :geschlecht (s)  :typ (Ss, Pi) }
{ zeichensatz :sub  :stamm zeichens@atz  :typ (Ss, PUe) }
```

The first part of the entry is the word name, which must be the base form of the noun. :sub (or alternatively :substantiv) is the key word that assigns the part-of-speech.

The keyword :typ precedes the list of inflectional types of the noun. The available types are S, Ss, Sn, P, Pe, Pn, Pen, Per, Ps, Pss, Pi, Pue, PU, PUe, PUen, PUer. The S-types describe how the genitive singular is formed from the base form: either by appending nothing (S), as in die Adresse → der Adresse, by appending s or es (Ss), as in das Bild → des Bild(e)s, or by appending n or en (Sn), as in der Mensch → des Menschen. From this S-type assignment all singular noun forms can be (and are) entered into the dictionary (nominative, genitive, dative, and accusative case).

The P-types describe how the nominative plural is formed from the base form:
for P/Pe/Pn/Pen/Per/Ps by appending nothing/e/n/en/er/s,
for PU/PUe/PUen/PUer by turning the first or @–marked vowel into the corresponding Umlaut *plus* appending nothing/e/en/er[5],
for Pss by turning an ending s into sse,
for Pi by turning the latin ending o or us into i, and
for Pue by turning the latin ending um into en.

All nouns with singular type S are automatically assumed to be female, all others are assumed to be male. These heuristics find the correct gender for about 80 percent of all nouns. For the rest, gender can be announced explicitly with the keyword :geschlecht followed by a parenthesized list (!) of the letters m (for male), w or f (for female), and s or n for neutral. Multiple genders can be assigned to a single word.

If the noun is irregular, it may be necessary to assign different stems for singular and plural forms. How to handle this case is shown with the example algorithmus above.

[5] e.g. der Zeichensatz, die Zeichensätze

A.1.3 Adjectives

Entries for adjectives may look like

```
{ absolut   :adj }
{ bedeutend :adj :steigerungsstaemme (bedeutend, -,
                                      (bedeutenst, bedeutendst)) }
{ public    :adj :ungebeugt }
```

The first part of the entry is the word name, which must be the base form of the adjective. `:adj` (or alternatively `:adjektiv`) is the key word that assigns the part-of-speech. If the comparation endings are not `er`, `est`, the complete base forms for positive, comparative, and superlative can be given as a list of words (or word lists for alternative forms) after the keyword `:steigerungsstaemme`. It is also possible to specify that the word should be considered to be an adjective in an input sentence even if it appears without a inflectional ending by giving the keyword `:ungebeugt` in the entry last. This is needed to handle german usage of english adjectives.

Inflected adjectives are analyzed algorithmically and are not put into the dictionary as full forms. All adjectives are automatically also available as adverbs.

A.2 SARA role table

The role table associates the names of syntactic roles with a set of grammatical constructions and a set of question phrases that can be used to refer to this syntactic role. The standard set of syntactic roles resides in the file `saralib/sara.std` and has 33 entries. Although it does not need to be changed very often, its format is described here.

A very general example of a role table entry is the following:

```
<< agent "das Subjekt"
    :nur_aktiv
      :nominativ          ;'ich' schlage keinen Hund
    :nur_passiv
      :dativ von          ;'von mir' wird kein Hund geschlagen
      :dativ vom          ;'vom Nachbarn' werden alle Hunde geschlagen
      :akkusativ durch    ;'durch mich' werden keine Hunde geschlagen
    :nur_nominal
      :genitiv            ;das Klagen 'des Nachbarn'
      :dativ von          ;das Klagen 'von mir'
      :dativ vom          ;das Klagen 'vom Nachbarn'
      :akkusativ durch    ;das Klagen 'durch den Nachbarn'
  :fragen
    :nur_aktiv
      :frage (wer)        ;'wer' fragt mich
      :frage (was)        ;'was' krabbelt meinen Ruecken hinauf
    :nur_passiv
      :frage (von wem)    ;'von wem' werde ich gefragt
      :frage (von was)
      :frage (durch wen)
      :frage (durch was)
      :frage (wo)         ;(sinnvoll, wenn die Agenten SW-Objekte sind.)
>>
```

This entry can be read as follows: `agent` is the name of the entry (as to be used in `AT-DOMAIN` and `AT-RANGE` annotations). The following string is merely a free form description of the entry. All of the following is optional, except the keyword `:fragen`.

`:nur_aktiv` means "the following appearance forms are valid for verbal phrases (i.e. clauses) in active voice only". `:nominativ` means "one possible appearance of the `agent` role is a noun in pure nominative case (i.e. without a preposition)". The part of the line after the semicolon is a comment and gives an example for this appearance form.

`:nur_passiv` means "the following appearance forms are valid for verbal phrases in passive voice only". `:dativ von` means "one possible appearance of the `agent` role is a noun in dative case preceded by the preposition `von`". Again, the part of the line after the semicolon is a comment and gives an example for this appearance form. The explanations for the other appearance entries are analogous.

`:nur_nominal` means "the following appearance forms are valid for nominal phrases only". There could have been appearance cases before the `:nur_aktiv` keyword. This had meant that they should be valid for all situations, i.e. active and passive verbal phrases as well as nominal phrases.

The keyword `:fragen` is (non-optionally) needed for purely syntactic reasons, to ease parsing the entry.

`:frage (wer)` means "the phrase `wer` at the beginning of a sentence indicates that the sentence maybe asks for the `agent` role." Arbitrarily many question words may be given and must all appear in exactly the given order to match the appearance form.

`:nur_nominal` is not possible (nor sensible) in the `:fragen` section of the entry.

Question forms which contain a noun, such as Welche Funktion are implicitly derived from all the appearance forms from above the `:fragen` keyword, that are valid for verbal phrases.

B List of Generated Case Frames

These are the case frames that are generated from the example modeling on page 11. The listing format is that of YAKR itself:

```
Call-Actions  [C-Call-Actions]  (
  agent has-agent (C-Functions),
  thema calls (C-Functions),
  attribut with (C-Parameters),
  benennung has-synonym (C-Call-Actions))
anything  [C-anything]  (
  benennung has-synonym (C-anything))
reads  [R-reads]  (
  agent DR (C-Input-Functions),
  thema RR (C-Data-Objects))
specifiers  [C-specifiers]  (
  benennung has-synonym (C-specifiers))
Parameters  [C-Parameters]  (
  gen_von INV(has-Parameter) (C-Functions),
  benennung has-synonym (C-Parameters))
Functions  [C-Functions]  (
  benennung has-synonym (C-Functions))
sein2  [W-sein]  (
```

```
  adj_adv ?? (C-anything) (21),
  agent ?? (C-anything) (21))
Data-Objects  [C-Data-Objects]  (
  benennung has-synonym (C-Data-Objects),
  gen_von INV(has-Parameter) (C-Functions))
has-Parameter_HAOd  [R-has-Parameter]  (
  agent DR (C-Functions) (21),
  thema RR (R-has-Parameter) (21))
has-Parameter_HAOc  [R-has-Parameter]  (
  agent RR (R-has-Parameter) (21),
  definition RR (C-Data-Objects) (21))
has-Parameter_HAOa  [R-has-Parameter] (rollensynonym) (
  benennung no-KRS-role (C-Data-Objects),
  gen_von INV(has-Parameter) (C-Functions))
Files  [C-Files]  (
  gen_von INV(has-Parameter) (C-Functions),
  benennung has-synonym (C-Files))
sein1  [W-sein]  (
  definition ?? (C-anything) (21),
  agent ?? (C-anything) (21))
Objects  [C-Objects]  (
  benennung has-synonym (C-Objects))
GF  [C-anything]  (
  definition no-KRS-role (C-anything) (21),
  agent no-KRS-role (C-anything) (21))
Input-Functions  [C-Input-Functions]  (
  benennung has-synonym (C-Input-Functions))
Output-Functions  [C-Output-Functions]  (
  benennung has-synonym (C-Output-Functions))
haben1  [W-haben]  (
  thema ?? (C-anything) (21),
  agent ?? (C-anything) (21))
writes  [R-writes]  (
  agent DR (C-Output-Functions),
  thema RR (C-Data-Objects))
Lines  [C-Lines]  (
  gen_von INV(has-Parameter) (C-Functions),
  benennung has-synonym (C-Lines))
actions  [C-actions]  (
  benennung has-synonym (C-actions),
  agent has-agent (C-Functions))
Characters  [C-Characters]  (
  gen_von INV(has-Parameter) (C-Functions),
  benennung has-synonym (C-Characters))

Fallschablonen zu 'sein' : (GF, has-Parameter_HAOc)
Fallschablonen zu 'haben': (has-Parameter_HAOd)
```

sein1, sein2, and haben1 are the case frames that are used internally to parse all inputs with sein or haben as main verb. Instantiations of these case frames are then converted into instantiations of the appropriate _HAOc, _HAOd, _HAc, and _HAd case frames by a unification algorithm. This method is used to avoid the extremely long running time of the parser that would result if all sein/haben case frames were directly used for parsing. GF is the so called *general frame* that is used to parse inputs of the form Welche *A* sind *B* where both A and B are object concepts.

C User interface commands

The help page of the SARA command interpreter looks like this:

```
q   - Quit. Beende die Sitzung
s   - Rufe eine Shell auf
l   - Lese SARA-Wissensbasis (kann #krsinclude enthalten)
L   - Lese reine YAKS-Wissensbasis
w,k - Zeige Woerterbucheintraege/Konzepte
r,f - Zeige Rollen/Fallschablonen
t   - Teste die Wissensbasis auf Konsistenz
F   - Erzeuge die Fallschablonen
a   - Anfragen anzeigen (ein/aus)
c,i - Charts/Instanziierungen anzeigen (ein/aus)
d   - Frage nach Ausgabedateinamen bei w,k,K,r,f,t (ein/aus)
D   - Dateinamen aendern
C,I - Charts/Instanziierungen auf EDGE-Datei schreiben (ein/aus)
H   - Konzepthierarchie einmal auf EDGE-Datei schreiben
e   - Lenke die Standardeingabe um
!   - Dump der Wissensbasis nach |less
.   - wiederhole letzten Eingabesatz
,   - YAKS-Kommandointerpretierer aufrufen (Verlassen mit 'quit')
<   - Einlesen des binaeren Woerterbuches aus ~/tmp/sara.wb.bin.Z
```

Some of these commands are described now.

l prompts for a filename and reads that file as a SARA knowledge file, which may contain (a) dictionary entries, (b) role table entries, (c) `#include` of other SARA files, and (d) `#krsinclude` of YAKS files. A dash can be given as a file name and denotes the standard input.

L prompts for a filename and reads that file as a YAKS knowledge base file.

w,k,r,f prompt for a string x and display, respectively, the corresponding dictionary entries, concepts, role table entries, or case frames from the SARA knowledge base. The output should be more or less self explanatory. For `k`, the names in the output are prefixed by either `R-, C-` or `I-` to display that this SARA concept is a role, concept, or instance, respectively, in YAKS. Role concepts are further prefixed by either `Sy_, Th_, HA_, HAO_`, or `?_` to display that the role is a synonymless, theme, has-attribute, has-action-or-object, or other role, respectively. Concept concepts may be further prefixed by either `AO_, A_`, or `?_` to display that the concept is an action-or-object, attribute, or other concept. If the string that is input at the prompt starts with a question mark, all objects whose names contain the following string as a substring are displayed. If the string is completely empty (i.e. just a <Return>), all entries are displayed.

"<" reads the binary form of the dictionary from a file whose name is given in the file `.sararc`; this is very much faster than to parse the source form of the dictionary, but is only possible as long as the dictionary is completely empty. This is usually the very first command issued in a session.

">" writes the complete dictionary onto a file whose name is also given in the file `.sararc`, for later use with the command <. To avoid accidents, it is clever not to have the same name in `.sararc` for the output file as for the input file.

D Example Session

This section contains a short example session with YAKR, using the example knowledge base from above. YAKR's output is in `this typeface` while user input is in *this typeface*. Comments are indented.

```
Willkommen bei YAKR          ? fuer Hilfe
```

```
SARA: <
lese sara.wb.bin.Z : sis.wb...../.....................
```

Now the SARA dictionary has been read in.

```
SARA: l
Lese SARA-Wissensbasis ein.
Dateiname der SARA-Wissensbasis:  bsp
```

Remark: The contents of the file `bsp` are

```
#include    "sara.std"
#krsinclude "daten/bsp_yaks"
```

At this point a lot of output is generated that shows the names of the objects in the knowledge base as they are created while the knowledge base file is being read. This output is not shown here.

```
SARA: F
```

This command creates announces that all knowledge files have been read in and generates the case frames. Again, a lot of output not shown here is generated that shows the names of the case frames as they are created.

```
SARA: Funktionen, die eine Zeile ausgeben
```

```
fputs
```

```
SARA: welche Funktionen geben etwas aus
```

```
fputs  writes  string-s
fputc  writes  character-c
```

```
SARA: kann irgendeine Funktion ein Zeichen einlesen ?
```

```
fgetc reads character-c
```

```
SARA: Ausgabefunktionen
```

```
fputc     fputs
```

```
SARA: Funktionen, die ausgeben
```

```
fputc     fputs
```

```
SARA: i
Instanziierungen anzeigen wird EINgeschaltet
```

This command toggles the displaying of the instantiated case frames and the YAKS queries generated from them.

```
SARA: Funktionen, die ausgeben

"Funktionen, die ausgeben" -->

[:I L3 G4 C-Functions,
        [Gw Fngda Np P3 substantiv funktion]
  [:I L2 G24 R-writes, Relativsatz Np, P3, (aktiv,praesens,indikativ,Nebensatz)
    [:rolle L2 G2 agent/DR
      [:I L2 G2 C-Functions,
        [Gw Fna Np P3 substantiv funktion]]]]]

Q-CONCEPT ?cvar = AND(Functions
    SOME(writes)).  /* Concept */

fputc fputs

SARA: q

Tschuess !
```

E Possible Problems with Natural Language Interfaces

The following hypothetical dialog may serve to illustrate some of the difficulties that may arise in the use of intelligent systems with natural language interfaces. Actually, this projection was made in the year 1973, so the setting is with a slow printing terminal. Please take care to understand the initial instructions.

Hello, I am Your utility service computer. To provide the best possible service to You, my programmers have given me the capapbility to converse in basic English as well as the ability to learn.
Enter your requests from the keyboard.
Your requests must start with //
*To cancel your typein, type *** and press the CARRIAGE RETURN key.*
To interrupt my typeout, depress the MARGIN RELEASE key.
To correct a typein, backspace and type over the errorneous characters.
To enter a typein, type ! or ?.
To terminate my services, type "finished!".
You will be billed for the time used at the end of the month.
What is your credit card number ?

//372AC-2631AFX.!

What is your name ?

//J. Womsely Rogers III.
!
//!

Credit card number validated.
What may I serve you, Mr. Rogers ?

//Do you provide library reference services ?

*Yes. I have access to 1.27*10E+7 volumes of books and 3.26*10E+7 publishers of magazines.*

//1.27*10E+7?

*"1.27*10E+7?" meaning unclear. Please clarify.*

//Sorry. What does 1.27*10E+7 mean ?

*1.27*10E+7 is number. Other legal notations are 12700000 decimal or 110000110101000000000000 binary.*

//Oh, I see. Can you give me all your references on Tarot cards ?

Tarot cards ?

//Tarot cards !

Tarot cards ?

//Cards used by gypsies to foretell the future !

I have major subject indices for subjects "card", "gypsy", "gypsum", and "future". Which do you wish displayed ?

//I want to know about Tarot cards.
!
//!

Tarot cards ?

//OK! You win. I wa
"OK!" meaning unclear. Please clarify.

//Listen you goddam*INPUT CANCELLED***
//Please display your index for subject "Gypsy".
//!

Books or magazines ?

//Both!

Which first ?

//Books?

"Books?" meaning unclear. Please clarify.

//Books!

Do you wish the listings in random order, alphabetic order by title, alphabetic order by author, grouped by publisher, or odered by date of publication ?

//Whatever.
//!

"Whatever." meaning unclear. Please clarify.

//Present the data in whatever order you prefer!

*Do you wish the listings in random order, alphabetic order by title, alphab****TYPEOUT INTERRUPTED****

//By date of publication!

Earliest or latest first ?

//Latest!

First or last ?

//Eithe**INPUT CANCELLED***
//First
!
?
!
//!

Yes.

//Yes?

"Yes?" meaning unclear. Please clarify.

//"Yes" what ?

Yes, I can give you all my references on Tarot cards.

//Why the Hel**INPUT CANCELLED***
//Why didn't you say so in the first place ?

Why didn't I say what ?

//Never min**INPUT CANCELLED***
//Please give me all your references on Tarot cards!

*Do you wish the listings in random order, alphabetic order by ti****TYPEOUT INTERRUPTED****

//RANDOM ORDER
//!

Several references are listed as classified/restricted information. If you wish access to them, please enter access code.

//What's classified about Tarot cards ?

Tarot cards ?

//Tarot cards
//**INPUT CANCELLED***
//Omit the classified references.
//!

*Several references are listed as restricted information. If you wish acce****TYPEOUT INTERRUPTED****

Omit the restricted information.
!
//!

No input received. Please enter next request.

//Omit the restricted information!

Published hardbound and undated paperback references for subject "gypsum" follow. Dated paperback publications are included in maga`***TYPEOUT INTERRUPTED***`

//I don't want to know about Gypsum. I want to know about Tarot cards.
//Cancel
!
!
?
!
//!

"Cancel" meaning unclear. Please clarify.

//Finised!

Word "Finised" not in vocabulary tables. Please provide definition.

//Finished!

Request illegal at this time. Please provide definition of word "Finised".

//"finised", transitive verb, from the french "fini".
//Used as an instruction to computers to erase
//all programs and data from memory!

Definition recorded. Thank you. Enter next request.

//You're quite welcome. Finised!

```
@&‘\J\
8pF.pB4ZL^T\v0^x$n( Zd:‘lv0(TL<D|.XvfNT.6(t^^|4,,Jl|jll(^r>TDd
||B^NNx"XX(JxZ<L668<pz|.vtx*VL@xP,vZZ ZPDz JzD
X$PTHnjnbd6H428p6n:0Fn,D>4(.jr|P4<ljZ6rjR j,,Z<4hnn|@n<Rlx
v$L<$b(‘^xx.<<\ @x,Jn.((2T6dr"lFrTJNrR|OH|(r&OF\Rd@L48:FpJ
LB"2v
|:|"XxRZHt&^<DjpXd*n.znD4L<6‘ vfn
R|,VP.(2X>@‘D&R\v@pN4v8V&D8R8bJX‘p&V@vTV>Z v:*fOL$V‘x\x2
<&d,$|ld‘8$T|TDr>vXl\D$6D\bTn4:,2tvhlR
8pVh(^@X"vD8 bPdJP"",fphXl
p6fxB6V.>h!!!!!!!!!!&h*O*(.&"<,(N4>.L8‘nFd
$Hz88rlODBF:H84z6$VpBjlNrJt@ZB&lbJ(*LJO
VddfR$Tz|N&HX&2<hfNrt.‘L\ ZhD ^D@:Z\zBJPt6,:
Ln@Z6\nLLB<O&*LL&B"8Xz.‘*BB.,z(f$x&tj@ONxTPhTDdp4Xp,n.&
PDLj2P||.LJ&z <^.D8$^dL^l^DRZ$2\ zf8z^^>B8>bv,
\Zbrn>V4r:2r4\OLn D.lL^RJL@D>"2H8RTPd*,"VJ&2
"bttR,xx2|d@LxTF rzp????V$Lh‘ PzLTNxzZdb>bf46bDN4|4‘ v
f**r,V2PT2v@dj|\jl"4(‘LLjLBDj4X: |:npPn60l\LnZ,4">
^Z$‘PvJFFp
4vbO|@Hv$6|‘f|tFjn2
$nbj‘tLZrzjLJPjXtpbbNVh4JvjzlD"NVH&bld
^,|*LXX&L2^.ht
 <h"hP,\Hr.|&DX6$lzV@h4v\t6$>
>$Hll(.P lD\X88F2,pj68pXd.L ^JP(tFB\jn(vxT v|V:p|F
t42vH2<‘86Z""tv*Lrpd‘r>2dzZ
```

```
z|4Tn:DxTNFTH.VrLB(J22tnbZ‘pOZ*Vr@\tlPFRPB&6b2bR|Z
V(&&x^b:h&‘L(J8XZ*:N\ NdpX(< vnNZD.,FFfN
F8O&>XXNJj?
```

(From: DATAMATION, April 1973, pp 72–73, by Donald Kenney)

For full enjoyment, now start again and read *carefully.*

References

[1] Rolf Adams. KRS — a hybrid system for representing knowledge in knowledge–based help systems. In *[21]*, pages 129–133, 1991.

[2] Rolf Adams. *Wissensrepräsentation und -akquisition in einem natürlichsprachlichen Software-Informationssystem.* PhD thesis, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, November 1992. to appear.

[3] Emmon Bach and Robert T. Harms, editors. *Universals in Linguistic Theory.* Holt, Rineheart, and Winston, Inc, London, Reprint 1972.

[4] Bruce Ballard. A lexical, syntactic, and semantic framework for a user-customized natural language question-answering system. In Martha Evens, editor, *Lexical-Semantic Relational Models*, pages 211–236. Cambridge University Press, 1988.

[5] M. Bates and R. Bobrow. *Natural Language Interfaces: What's Here, What's Coming, and Who Needs It*, chapter 10, pages 179–194. Ablex Publishing Company, 1984.

[6] Ted Biggerstaff and Charles Richter. Reusability framework, assessment, and directions. *IEEE Software*, 4(2):41–49, March 1987. also in [36] and in [7].

[7] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability — Concepts and Models, Volume I.* ACM Press Frontier Series. Addison-Wesley, Reading, Mass., 1989.

[8] R.J. Brachman and J.G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

[9] Finn Dag Buø. A natural language interface for KRS. Master's thesis, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, D-7500 Karlsruhe, Germany, 1991.

[10] J. Carbonell. Requirements for robust natural language interfaces: The Language Craft and XCALIBUR experiences. In *International Conference on Computational Linguistics*, pages 162–163. Association for Computational Linguistics, 1986.

[11] J. Carbonell and P. Hayes. Coping with extragrammaticality. In *International Conference on Computational Linguistics*, pages 437–443, 1984.

[12] J. Carbonell and P. Hayes. Natural-language understanding. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1, pages 660–677. John Wiley & Sons, 1987.

[13] Jaime G. Carbonell and Philip J. Hayes. Robust parsing using multiple construction-specific strategies. In *Proceedings of the Seventh IJCAI*, pages 432–439, August 1981.

[14] P. Devanbu, R.J. Brachman, P.G. Selfridge, and B.W. Ballard. LaSSIE: A knowledge-based software information system. *Communications of the ACM*, 34(5):34–49, May 1991.

[15] P. Devanbu, P. Selfridge, B. Ballard, and R. Brachman. A knowledge based software information system. In *International Joint Conference on Artificial Intelligence*, pages 110–115. Morgan Kaufmann Publishers, 1989.

[16] Charles J. Fillmore. The case for case. In *[3]*. Holt, Rineheart, and Winston, Inc, 1968.

[17] Martin Fischer. Realisierung eines Wissensrepräsentationssystems für die Repräsentationssprache KRS. Master's thesis, Universität Karlsruhe, D-7500 Karlsruhe, 1991.

[18] G.W. Furnas, T.K. Landauer, L.M. Gomez, and S.T. Dumas. The vocabulary problem in human–system communication. *Communications of the ACM*, 30(11):964–971, November 1987.

[19] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. Wiley, Chichester, 1991.

[20] B. Grosz, D. Appelt, P. Martin, and F. Pereira. TEAM: An experiment in the design of transportable natural language interfaces. *Artificial Intelligence*, 32(1987):173–243, 1987.

[21] IEEE Computer Society. *The Seventh Conference on Artificial Intelligence Applications*, Miami Beach, Florida, February 1991. IEEE Computer Society Press.

[22] Michael Loren Mauldin. *Information Retrieval by Text Skimming*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1987. also as Techreport CMU-CS-89-193.

[23] Andy Podgurski and Lynn Pierce. Behavior sampling: A technique for automated retrieval of reusable components. In *Proceedings of the 14th International Conference on Software Engineering*, pages 349–360, May 1992.

[24] Lutz Prechelt. Ein Fallschablonenzerteiler für Deutsch. Master's thesis, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, D-7500 Karlsruhe, Germany, 1989.

[25] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, January 1987.

[26] Ruben Prieto-Diaz. *A Software Classification Scheme*. PhD thesis, Department of Information and Computer Science, University of California, Irvine, CA, 1985.

[27] Ruben Prieto-Diaz. Classficiation of reusable modules. In *in [7]*, pages 99–123. ACM press, 1987.

[28] Ruben Prieto-Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):89–97, May 1991.

[29] M. Ratcliffe. Report on a workshop on software reuse held at Hereford, UK, on 1-2 may, 1986. *ACM SIGSOFT Software Engineering Notes*, 12(1):42–47, January 1987. also in [36].

[30] Eugene J. Rollins and Jeannette M. Wing. Specifications as search keys for software libraries. In *Proceedings of the International Conference on Logic Programming*, Paris, June 1991.

[31] R.C. Schank. Conceptual Dependency: A theory of natural language understanding. *Cognitive Psychology*, pages 552–631, 1972.

[32] Peter G. Selfridge. Knowledge representation support for a software information system. In *[21]*, pages 134–140, 1991.

[33] S. Shieber, editor. *An Introduction to Unification-Based Approaches to Grammar*. CSLI Lecture Notes 4. Chicago University Press, 1986.

[34] Tore Syvertsen. CPPREF — an information abstractor for C++. Master's thesis, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, D-7500 Karlsruhe, Germany, April 1992.

[35] Will Tracz. Software reuse: Motivators and inhibitors. In *Proceedings of COMPCON S'87*, pages 358–363. IEEE, IEEE Computer Society Press, February 1987. also in [36].

[36] Will Tracz, editor. *Software Reuse: Emerging Technology*. Tutorial. IEEE Computer Society Press, Washington, D.C., 1988.

[37] D. Warren and F. Pereira. An efficient easily adaptable system for interpreting natural language queries. *American Journal of Computational Linguistics*, 8(3-4):110–122, July-December 1982.

[38] Patrick Henry Winston. *Artificial Intelligence*. Addison Wesley, 1981.

[39] Murray Wood and Ian Sommerville. An information retrieval system for software components. *Software Engineering Journal*, pages 198–212, September 1988.

[40] Scott N. Woodfield, David W. Embley, and Del T. Scott. Can programmers reuse software. *IEEE Software*, 4(4):52–59, July 1987. also in [36].