Artificial Intelligence Review **14:** 253–275, 2000.
© 2000 *Kluwer Academic Publishers. Printed in the Netherlands.*

What Do You Know about Mail? Knowledge Representation in the SINIX Consultant

CHRISTEL KEMKE
Universität Bielefeld, Technische Fakultät, AG Neuroinformatik, Postfach 100131, D-33501 Bielefeld (E-mail: ckemke@techfak.uni-bielefeld.de)

Abstract. The SINIX Consultant is an intelligent help system for the SINIX operating system which answers natural language questions about SINIX concepts and commands and gives unsolicited advice to a user as well. In this paper the representation of domain knowledge in the SINIX Consultant will be discussed, i.e. the representation of concepts of the SINIX operating system in the so-called SINIX Knowledge Base. The SINIX Knowledge Base is a taxonomical hierarchy of SINIX concepts which are divided into objects and actions operating on these objects. A single concept in the knowledge base is described by a set of attributes reflecting structural or syntactical features, the use, application and purpose of the command or object, and additional information for explaining the concept to the user. The description of commands in the SINIX Knowledge Base which is mainly used in order to generate tutorial explanations and advice for the user, and the main ideas of describing SINIX objects are outlined in this paper.

Keywords: help systems, intelligent user interfaces, natural language processing, tutoring, user modelling, Unix

1. Introduction

The SINIX[1] Consultant (SC) is an intelligent help system for the SINIX operating system which answers natural language questions about SINIX concepts and commands and occasionally gives unsolicited advice to a user.[2] Example user questions are “How do I delete a file?” and “What is a directory?” and respective SC answers could be “You can use the command *rm* with the syntax *rm <filename>* to delete a file.” and “A directory contains files and/or other directories. It is possible to build a hierarchical structure of directories in SINIX. See also the related commands *mkdir* and *rm -r*.” The formulation of answers in SC depends on information from a user model which is built up during the dialog with the actual user, based on prototypical information; unsolicited advice is given in case the user enters ‘sub-optimal’ command sequences, i.e. too many commands to achieve a certain goal.

The *SINIX Knowledge Base* as part of the SINIX Consultant provides the information necessary to produce such answers and advice for the user. The

SINIX Knowledge Base (SINIX-KB) is a taxonomical hierarchy of SINIX concepts which contains descriptions of more or less abstract objects and actions of the SINIX domain. Domain concepts in the SINIX Knowledge Base on the basic level are commands and objects of the SINIX system, for example the generic objects ‘file’ and ‘directory’ and commands like ‘ls’. Higher level concepts represent more general and unspecific actions and objects, like ‘communicate-with-user’; they reflect a general user’s view of the system and correspond to natural language terms or mental model entities. A single concept in the knowledge base is described with respect to its function, structure, use, and relationships to other concepts. The main emphasis of the developed SINIX-KB was on the representation of commands, including characterizations of their application and function used for user-adequate explanations. SINIX actions are described by a set of attributes reflecting their syntactical structure, their function, and various aspects of their use and relations to other commands. These attributes are needed in order to generate tutorial explanations with a varying degree of detail based on the individual user’s state of knowledge regarding the SINIX system. The descriptions of commands given in the SINIX Knowledge Base are mainly used in order to generate tutorial explanations and advice. Besides the description of commands via a set of informal attributes, a formal representation of the semantical structure of commands and its potential use in question evaluation and planning processes will be outlined in this article. We will also describe the main aspects of the representation of objects in the SINIX Consultant.

2. Survey of the SINIX Knowledge Base

The construction of the SINIX Knowledge Base – the *SINIX-KB* – is based on ideas of the KL-ONE representation formalism (see Brachman and Schmolze 1984) but stresses the importance of the definition of semantic primitives, i.e. concepts and their attributes (or roles) used to provide tutorial explanations of domain concepts in response to a user’s question.

The current SINIX-KB is organized as a conceptual hierarchy in which the major differentiation is between objects and actions (see Figure 1).[3] The main emphasis in developing the SINIX-KB was placed upon the representation of SINIX commands. Actions in the SINIX-KB are comprised of concrete SINIX commands as well as actions which tend to be more abstract and can be realized through different SINIX commands. These *concrete* and *abstract commands* are described by attributes characterizing their syntactical structure, their semantics, their use, and relationships to other commands. Objects in the SINIX-KB, which are also arranged hierarchically, serve mainly as

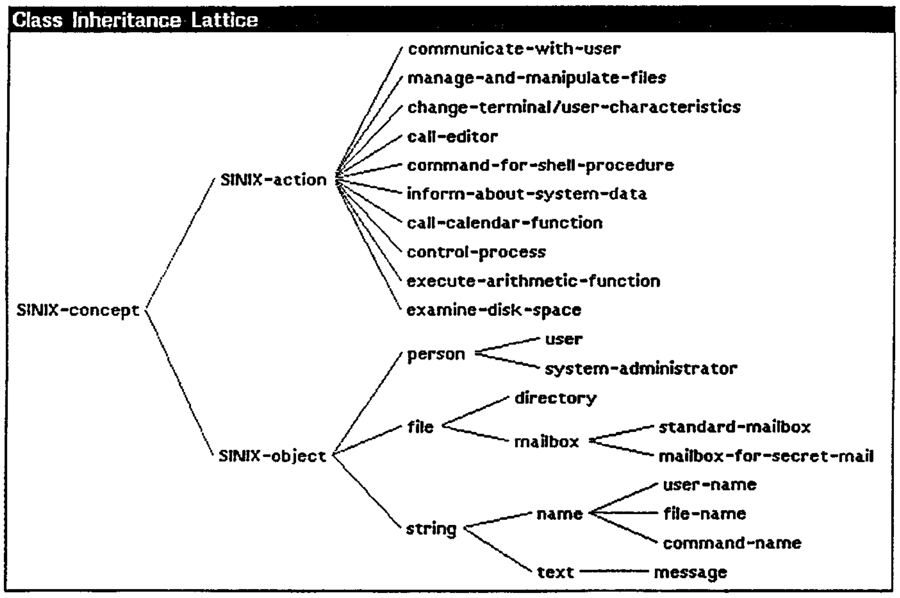

Figure 1. Structure of the SINIX knowledge base.

parameters for actions and thus provide the “material basis” for the system’s operations. They are described by attributes comprised of features representing their state, their structure and use.[4] The hierarchical arrangement of SINIX concepts is mainly used to determine a command (concrete or abstract) the user had specified in her question (see section 3.1). The outlined description of commands serves mainly to provide tutorial explanations based on the user’s expertise concerning the SINIX system (see section 3.2).

3. The Representation of Commands in the SINIX KB

3.1. *The action hierarchy*

The action part of the current SINIX-KB consists of about 300 concepts. About 180 are completely described by attributes, 90 of them being concrete SINIX commands and 50 abstract commands. In order to reduce the complexity of the representation task, we excluded shell programming as well as system-management commands.

Figure 2 shows a part of the action hierarchy which can be divided into several levels. Higher level concepts reflect more general actions or objects which, in turn, correspond to less specific natural language terms and reflect the user’s general mental model of the SINIX system.

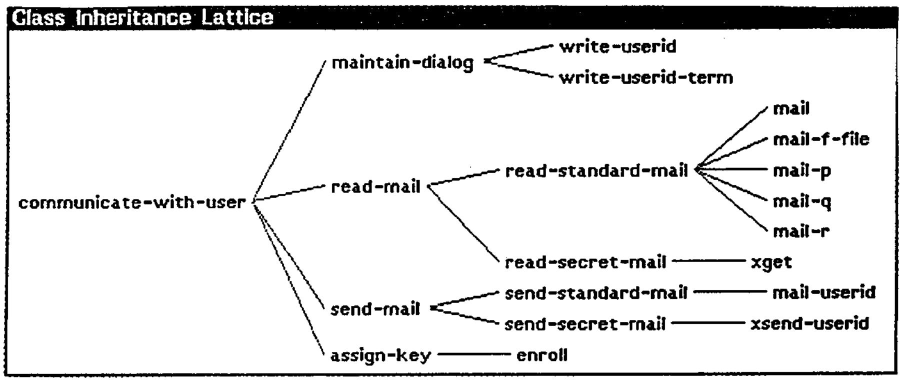

Figure 2. A part of the hierarchy of SINIX actions.

Direct subconcepts of the concept *SINIX-action* (see Figure 2) reflect the division of commands into task areas and are therefore called *themes*, e.g. the concept *communicate-with-user*.[5] Descriptions of actions at this global level are useful in answering nonspecific user questions. These arise if the user is unfamiliar with the system but at least recognizes its principal features and capabilities. The concepts on the level below correspond to "canonical" natural language terms – mainly verbs and, occasionally, objects and modifiers used for naming the subordinate commands. Concepts at this level are called *abstract commands*. The leaves of the hierarchy represent *concrete SINIX commands* determined by a command name and, occasionally, an option and one or more parameter variables.

In some cases, one and only one concrete SINIX command exists which realizes an abstract command denoted by a canonical natural language term or which corresponds to a mental model entity, respectively, e.g. the *enroll* command, which assigns a key for receiving secret mail. In this case, there is only one level of abstract commands. In other cases, different command names exist, e.g. *mail* and *xsend* for sending mail. In these instances, a second level of abstract commands had to be introduced. This arrangement of commands is used by the Question Evaluator in determining the most specific concept which is addressed in the user's question and whose description serves to answer it. The Question Evaluator finds this concept using the (canonical) verb representing the action, the object affected, and, occasionally, attributes and modifiers mentioned in the phrase. The Answer Formulator makes use of the hierarchical arrangement by inferring from the concept's position in the hierarchy which information is available for explaining the concept.

3.2. *The description of command concepts*

Concepts pertaining to the SINIX-KB are frame-like structures with a fixed set of *attributes* (slots) the values (fillers) of which are

- *concept names* representing relations between concepts,
- *syntactical structures* in the case of descriptions which are constructed using other attributes, e.g. the syntax of a command, and
- *non-analyzable strings or texts*, which are processed by other system components or are directly displayed to the user, for example, a complete description text or a reference to a bitmap.

The attributes describing SINIX actions (cf. Figure 3) can be classified according to three categories reflecting

- *syntactical* aspects,
- *semantical* aspects, and
- *pragmatical/tutorial* aspects

of the command. This is not a strict subdivision because some attributes may be used for different purposes, though the main reason for introducing them arises out of the classification.

Syntactical aspects

The attributes pertaining to this class are used to describe the syntactical structure of a command; they are *command name, parameters, option, syntax, sample command*. The 'sample command' is an instantation of a concrete command, i.e. a command with instantiated parameters and, occasionally, an option.

Semantical aspects

The attributes used for the semantical description include a *description text*, used in a first version of SC in order to explain the function of the command verbally, and a *graphical representation* illustrating this function. The graphical representation – as well as the sample command – is selected by the Answer Formulator when explaining a command to a novice user. Visual impressions should help a beginner to construct a rough mental model of the system's functions. The formal representation of the semantics of commands which is thought to be described by attributes like 'preconditions', 'main effect', and 'side effects' will be outlined in section 4.2 (see also Figure 5).

Pragmatical and tutorial aspects

We introduced several attributes to describe pragmatical or tutorial aspects of a command. These attributes are used to provide *preventive information* or *mnemonic help*, and they are used in *focussing* a command being sought or *learning* new commands not directly addressed in a question.

communicate-with-user

supers	SINIX-action
parameter	<parameter1>* user-id standard-file
	{<parameter2>} terminal
option	{<option>}
synopsis	<commandname> {<option>} <parameter1>* {<parameter2>}
description text	The concept ‘communicate-with-user’ refers to all commands within the SINIX system which are related to the communication between users.
possible-commands	mail xsend xget write enroll
important-commands	mail write
standard-command	mail
sample-command	mail guest

send-standard-mail

supers	communicate-with-user
command-name	mail
parameter	<recipient> user-id
option	nil
synopsis	mail <recipient>
description-text	The SINIX operating system allows you to send mail to another user. You have to specify the recipient of the mail by her user-id.
presupposed-concepts	user-id
sample-command	mail guest
standard-command	mail
appertaining-commands	mesg who mail
similar-commands	write xsend
subcommands	((END (CR . CR)))

mail<user>

supers	send-standard-mail
command-name	mail
parameter	<recipient> user-id
option	nil
synopsis	mail <recipient>
description-text	. . . send mail to another user. The mail is stored in her *mailbox*.
presupposed-concepts	user-id
sample-command	mail guest
similar-commands	write xsend
subcommands	((END (CR . CR)))
graphical-representation	BITMAP# . . .

Figure 3. Representation of SINIX action concepts.

Preventive information is information given to the user in order to prevent foreseeable errors or situations in which the consultant is not able to help. If the user asks, for example, for a command which causes a mode change – e.g. the *mail*-command or the call of an editor – SC informs her about important commands of the subsystem, so-called *subcommands*, e.g. the keys for ‘help’ and ‘quit’. It also refers to *presupposed concepts*, i.e. concepts the user has to know in order to understand and use a command correctly, if it is assumed that she is not familiar with them.[6]

Several attributes are used for *focussing* on a command the user is seeking for. These attributes are only defined for higher-level concepts, i.e. themes and abstract commands. *Possible commands* contain the whole set of commands of a certain task area and thus should include the one the user is looking for. With a novice, SC will only mention the *standard command*, i.e. the most frequently used command pertaining to a specific area, or *important commands*, i.e. a set of common commands which realize different abstract commands.

Mnemonic help is provided by referring to *similar commands*, i.e. commands having an effect similar to the one she asked for but which are familiar to her. Mnemonic help is also provided when suggesting several possible commands or important commands as described above.

Experienced users may be pleased to learn some new commands which are related to the task they are performing. Commands related to the respective command are the so-called *see-also-commands* also mentioned in manuals. The attributes ‘possible commands’ and ‘important commands’ may also serve to extend the user’s knowledge by providing her with a survey of available commands within a certain task area.

Complete descriptions of the theme *communicate-with-user*, the abstract command *send-standard-mail* and the concrete command *mail<user>* are shown in Figure 3.

4. Towards a Formal Semantics of Commands

The description of commands developed so far is appropriate in order to construct tutorial explanations but does not seem sufficient in order to provide the basis for sophisticated reasoning processes. A formal representation of the semantics of commands, i.e. mainly their effects, is necessary to enable problem solving, planning and other inference or reasoning processes needed in order to help a user perform her task. Thus, I developed the basis of a formal representation of the semantics of commands, which can easily be transferred to other operating systems and can also be extended to represent general command languages as well. The main ideas within the semantical representation are a classification of commands leading to a hierarchy of generic, primitive commands and a formal logical description of their function. Attributes describing the semantics comprise references to the objects affected by the command as well as preconditions and effects of their use. Since these attributes are inherited from higher-level generic commands, attributes describing modifications of these generic operations may be added at lower levels, finally resulting in descriptions of concrete SINIX commands.

This should lead to a functional description of commands integrated into a taxonomical or inheritance hierarchy as proposed by the KL-ONE knowledge representation system.

4.1. *A taxonomy of generic commands*

The central idea behind the semantical representation is to describe the effects of commands by using a set of “basic” or “generic” actions. Generic actions are also thought of as being canonical verbs, i.e. representatives of verb classes, and thus provide an interface to the natural language processing components.

The semantics of these actions is basically defined in terms of a state change of objects or, more precisely, an assignment of values to attributes of these objects, which again are described as concepts in the SINIX-KB. The semantics of entire SINIX commands should be derived from the semantical descriptions of these basic actions by specialisation and composition. Thus, I established a *taxonomical hierarchy of commands* which is intended to cover almost the entire set of SINIX commands and other command languages as well (see also Kemke 1987). The classification of operating-system or general software-system commands leading to this taxonomy was influenced by an empirical study of J. Carter (1986). In this investigation, people with varying degrees of expertise in computer systems had to designate operations used to perform a given circumscribed task. Actually, they tended to use only a few different traditional terms naming unspecific functions, e.g. *list* and *create*, which could be modified by adding object specifications etc. In contrast to Carter’s “taxonomy of user-oriented functions” the upper levels of which arise from a classification of commands with respect to *task areas*, our approach aimed at a classification oriented towards an *ontology of commands*.

The development of the taxonomical hierarchy depends mainly on a classification of commands with respect to

- the kind of effect caused by the action,
- objects and their attributes involved in the action, and
- variations of the action determined by modifiers.

The main part of the taxonomical hierarchy together with examples of classified SINIX commands is shown in Figure 4. The most basic categories of generic commands are *change* (change a system object), *inform* (inform about the state of the system), *construct* (create a new object from existing ones), and *transfer* (transfer an object). Other classification criteria involved, like objects, attributes etc., are shown in the far left column. Further specializations, for example by modifiers, are not completely covered in this table since they have not yet been developed in detail. Examples are the modifiers 'to' and 'from' used in some *transfer*-functions which determine the direction of the transfer, or the modification of transferring a file as part of a directory by specifying whether the original file is deleted afterwards (*move*) or not (*copy*). The middle row shows appropriate names of the respective generic commands determined on the basis of the considerations outlined above. Examples of SINIX commands belonging to the respective categories are contained in the third row. Note that some commands appear in several categories corresponding to different "views" of the same action. For example, the action of changing the name of a file yields in principle the same result as moving a file to another directory using the *move* command. A disadvantage of this taxonomy is that it doesn't provide a multi-dimensional classification which is, in some cases, more appropriate. For example, transfer operations are classified with respect to a location, e.g. *file*, and the direction of transfer, e.g. *write* to a file or *read* from it. An alternative approach is to define a classificatory scheme as a kind of multi-dimensional table, each dimension of which would correspond to a classification criterion. Entries would be generic or concrete commands, with certain entries being irregular or empty because these combinations of features are illegal or not available in the domain system. In this case the system would have explicit knowledge about concepts which are not realized in the domain, e.g. commands which don't exist but seem plausible to the user.

4.2. *Describing a generic command*

As outlined above, the representation of commands should comprise a formal description of their effects and preconditions in order to make reasoning processes possible, for example traditional planning methods, and access commands using *action* as well as *state* (*goal*) circumscriptions expressed in a user's statement or defined by a component of the SC system. Moreover, these descriptions should be integrated into a hierarchical representation of generic commands as described above.

The attributes used to represent the semantical description of commands are described in the following (see also Figure 5).

change		
∘ (system) variable	*set*	mesg, cd
∘ object		
– name	*rename*	mv
– feature value	*put*	chmod, stty
– contents	*edit*	vi, emacs
– exist-status		
∗ negative	*delete*	rm, kill
∗ positive	*create*	cat, enroll
∘ system/mode	*call*	mail, bc
inform		
∘ entities		
– contents	*list*	cat, ls
– feature value	*get*	file, ls
– location	*find*	whereis, find
– description	*help*	man, apropos
∘ relations		
– part-object	*search*	grep
– object-object	*compare*	cmp, diff
transfer		
∘ output-device		
– printer	*print*	lpr, cat
– screen	*display*	more, cat
∘ storage-device		
– file		
∗ to	*write*	cat
∗ from	*read*	cat, more
– directory		
∗ delete original	*move*	mv
∗ keep original	*copy*	cp
– tape/disk	*store*	far, tar
∘ user		
– mailbox		
∗ to	*send*	mail, xsend
∗ from	*receive*	mail, xget
– screen	*talk*	write, cat
∘ input-device		
– keyboard	*type*	–
construct		
∘ object		
– many-to-one	*combine*	join, cat
– one-to-many	*devide*	split
– one-to-one	*transform*	rev, prep
∘ value		
– arithmetics	*compute*	bc, units
– expressions	*evaluate*	expr, test

Figure 4. A taxonomy of generic commands.

move-files	
#args	*2*
arg-name#1	*fil1*
arg-type#1	*file*
arg-name#2	*fil2*
arg-type#2	*file*
trivial-precond	*fil1.exist-status = EXIST*
severe-precond	*read-access (actual-user, fil1)* $\wedge$ *write-access (actual-user, fil2)*
main-effect	*fil2.exist-status := EXIST*
	$\wedge$ *fil2.contents:=fil1.contents*
	$\wedge$ *fil2.creation-date:=fil1.creation-date*
side-effect	*fil2.location.#elements := fil2.location.#elements + 1*
	$\wedge$ *fil2.location.contents := fil2.location.contents* $\cup$ *fil2*
	$\wedge$ *fil1.location.#elements := fil1.location.#elements – 1*
	$\wedge$ *fil1.location.contents := fil1.location.contents* $\setminus$ *fil1*

Figure 5. Semantical description of a *move* function.

Arguments

First of all, attributes specifying the *arguments* of the command, i.e. the objects it operates on, have to be defined. Each argument is described by an internal, arbitrary name *arg-name* which serves as a variable or formal parameter of the command and may be instantiated with names of concrete objects given as actual parameters, for example during a planning process. In case of explanation processes where no reference to concrete objects occurs, these variables are not instantiated. The attribute *arg-type* refers to the type of the argument object, i.e. the concept or class of objects in the knowledge base it belongs to.[7] Attributes defined with respect to this class may be used additionally in the description. They are accessible using the type specification, for example the attribute *location* defined for objects being embedded into a surrounding structure, e.g. files, or the attribute *#elements* for sets, e.g. directories (see section 5).

Preconditions and effects

The representation of actions by defining pre- and postconditions is known from traditional planning systems. Usually, these systems have been applied to well-defined problems in very simple domains like the blocks world and the application of planning methods to more complex domains immediately leads to serious problems. With a technical system like SINIX, the domain is still simple and structured, though far more complex than a blocks world. Thus a *complete* description of pre- and postconditions does not seem appropriate in explaining a command to a user or applying planning algorithms.

For example, in most cases the precondition of a command operating on an object demands the existence of this object. This is not true, of course, for *create* commands. Thus the existence of an object used as an argument of a function is a "trivial precondition" and should therefore be handled differently in using reasoning and explanation processes rather than a "severe precondition" like having read access to a file in the case of the *copy* command. A similar problem arises in determining effects of a command. Usually, when selecting and applying a certain command, the user has one main effect of the command in mind resulting in a certain state of the system. But apart from this main effect, commands may have side effects which in general are of no interest to the user but in some cases may still be important.

Thus, I decided to differentiate between relevant and irrelevant preconditions and effects:[8] Preconditions are divided into *trivial* and *severe ones*, and effects are divided into *main effects* and *side effects*.

Preconditions are described as formulas of a many-sorted first order logic, where sorts correspond to concepts and are equivalent to one-place predicates. Effects are described using an additional value-assignment operation $v{:=}t$ where v denotes a variable and t a term.[9] This is in contrast to the formalization of postconditions in traditional planning methods. Nevertheless, the description of effects is also referred to by the term 'postcondition'.

Results

Instead of causing a state change of objects, some commands are primarily used to produce a *result*, for example the *construct* functions listed in the taxonomy. In these cases, it seems appropriate to introduce one or more attributes named *result* which refer to the generated structures, if they are not explicitly given as arguments. Results are also described according to their *type* and their *location*, e.g. standard output or a file. Some operations of UNIX-like command languages write by default to standard output, but often the result is moved to a file or used as input to another command. Thus the explicit declaration of the destination of the result is also useful in defining I/O-redirection and pipelining.

Modifiers

Other descriptive attributes of commands are *modifiers* which specify the mode of operation or a specific variant of a generic command. Modifiers for UNIX commands often but not always correspond to respective options. An example is the specialization of reading a sequence of objects, e.g. your actual mail, by specifying whether the sequence should be read in normal or reverse order, e.g. using *mail* or *mail -r*.

4.3. *An example*

In order to get an impression of an entire representational construction, refer to the description of the generic command *move* in Figure 5. The command has two arguments: the source and the destination object which in this case are both files located at directories.[10] The precondition states that the individual user communicating with SC, denoted by '*actual-user*', must have read access to the old file and write access to the new one.[11] The main effect of the command is the creation of a new object which has the same contents as the first one.[12] We also have to note the side-effect of the command with respect to the directory containing the new file, which has an additional element after the completion of the operation. This fact is stated explicitly as a side-effect.

4.4. *Specializing generic commands*

The complete set of attributes and their values should be inherited through the specialization paths defined in the hierarchy, thus reducing the complexity of representational structures. In order to obtain a representation system similar to KL-ONE, we have to define the meaning of a 'specialization of an action'.[13] Concerning attributes representing relations between concepts corresponding to roles in KL-ONE, a concept is specialized[14] by

- restricting the set of possible values of an inherited attribute
- the addition of attributes

Problems arise in defining a subsumption relation with respect to formulas describing preconditions and effects because these are usually not defined as concepts, i.e. sets of attribute values. In an initial approach to this problem, descriptions of subconcepts are derived by modifying the inherited information in the following ways:[15]

- The set of possible attribute values is restricted.
- Attributes are added.
- Propositions are added to an inherited formula by conjunction.
- One of several propositions connected by disjunction is deleted from an inherited formula.

Note that the first case subsumes specializing the type of an argument or, in other words, the concept the respective object belongs to.

Relating to this notion of specializing actions, we can describe, for example, *delete*-functions as shown in Figure 6. Note that the *delete*-command inherits the set difference operator '$\setminus$' from the set-object though the definition is written out in this example. The symbol '$\equiv$' denotes the equivalence of formulas or terms. Attributes not mentioned explicitly are inherited.

delete

superconcepts	*change-object-exist-status*
#args	*1*
arg-name#1	*obj1*
arg-type#1	*object*
trivial precond	*obj1.exist-status = EXIST*
precond	*delete-access (actual-user, obj1)*
main-effect	*obj1.exist-status := NIRVANA*
side-effect	$(\setminus$ *obj1.location obj1)* $\equiv$
	obj1.location.#elements := obj1.location.#elements – 1 $\wedge$
	obj1.location.contents := obj1.location.contents – obj1.name

delete-files

superconcepts	*delete*
arg-type#1	*file*

delete-directory

superconcepts	*delete*
arg-type#1	*directory*
preconds	– inherited – $\wedge$ *obj1.#elements = 0*

Figure 6. Inheritance of semantical descriptions for *delete* functions.

5. Describing Objects

The description of objects should be based on general structures, e.g. mathematical structures like *set, sequence, numbers* together with the usual operations and some access functions. These operations may be regarded as primitive and therefore can be defined using functions provided by the respective implementational language. Nevertheless, it would be more precise to reduce these functions again to basic value assignment operations.

The most fundamental structure is the *set* concept (see Figure 7) which is also used as a base for other structures, for example *sequence*. For the sake of simplicity, the symbols for set difference and union are also used for denoting the removal of an element from a set and the addition of an element to a set. INT_0^+ denotes the set of natural numbers including 0 and symbols for infinity and 'indefinite'. A term 'object*' denotes a set containing an arbitrary number of elements of the type 'object'. At the moment, these mathematical structures are intended to be general concepts which pass on their descriptions to subconcepts in the hierarchy of objects. Another way to describe structures of objects is to introduce an attribute 'structure' whose value is a structure-defining expression constructed from 'structure-building operators', for example the star-operator, concat-

set

superconcepts	*structure*
elements	*object* *
element type	*object*
#elements	INT_0^+
operations	$\cup$ A B *union of the sets A and B*
	$\setminus$ A B *difference of the sets A and B*

ordered-set

superconcepts	*set*
$<$ a b	*compares two elements a∈A and b∈B*

sequence

superconcepts	*structure*
base	*set*
element type	*object*
length	INT_0^+
operations	$\setminus A\ a$ *remove element a from sequence A*
	$\circ$ A B *concatenate sequences A and B*
access-functions	$n - th$ A *access* $n - th$ *element of sequence A*

Figure 7. Mathematical structures as basic objects.

enation, union etc., and concepts in the knowledge base, e.g. *word** for *text*.

Domain objects are described by several attributes which are relevant with respect to the use of these objects. It is important to differentiate between attributes which describe features of the object, e.g. length of a file, and attributes which are supplementarily introduced and describe other aspects of the object, e.g. its name or exist-status. The set of most common attributes used to describe objects together with the types of these attributes is given in Figure 8. Note that an additional attribute *identifier* has been introduced which, in case of individuals, contains a unique name of the individual object, e.g. the number of a process or the pathname of a file. The attribute *operations* provides a connection to the action branch and refers to generic actions or SINIX commands operating on the object. This relation is inverse to the argument relation of commands. *Restrictions* are on the one hand used to define absolute value restrictions for attributes, e.g. the length of a name. On the other hand, they can describe relations between attributes. Though not always adequate, the first case can also be handled by defining a new concept which contains the proper set of possible values.[16] Usually, an object has several *features* which are represented as separate attributes.

The definition of the concept *directory* is shown in Figure 9.

superconcepts: *concept* *	direct superconcepts of the object in the knowledge base
name: *name*	name of an object used to refer to it
identifier: *concept*	unique label which identifies an object
exist-status: *exist-states*	the state of existence of an object
location: *concept*	object which contains the respective object permanently or temporarily
operations: *action* *	actions operating on the object
purpose: *text*	verbal description of the purpose of the object
restrictions: *logical formula*	restrictions describing relations between attribute values and restrictions on possible values
<features> : *concept* *	features describing the object (listed separately for an individual object); features are again concepts in the knowledge base.

Figure 8. General description of an object.

Some remarks on attributes

A future prospect in the representation of objects is to integrate meta-knowledge. This knowledge should, for example, describe characteristics of attributes. In philosophy, a traditional classification of features is known, which, first of all, divides features into *essential* (*necessary*) and *nonessential* (*unnecessary*) ones; essential features are again divided into *constitutive* and *consecutive* ones. Consecutive features can be inferred from constitutive ones like, for example, the inferred attribute ‘possible commands’ mentioned in section 3.2. Constitutive features are comprised of *proper* and *common* ones. Features can be used as *modifiers* or *determiners*, but this differentiation often depends on their use in context. An important characteristic of features or attributes in this context is their behavior with respect to time. Features may be *temporary* (change dynamically without being influenced), e.g. the age of an object, they may be *changeable* by an action, e.g. the protection-mode of a file, or they may be *permanent* (unchangeable), e.g. the user-id.

It seems useful to represent some characteristics of attributes as meta-knowledge, for example, using ‘meta-objects’.[17] Meta-objects should also keep the information which attributes can be regarded as features of the object, perceived by the user as descriptive attributes (e.g. length of file) in contrast to attributes used to describe the objects internally (e.g. its name and exist-status).

directory

superconcepts:	*ordered-set*
elements:	*name* *
element-type:	*name*
name:	*name*
identifier:	*pathname*
exist-status:	{EXIST, NIRVANA, . . . }
location:	*directory*
operations:	*rmdir, chmod, ls, cd, . . .*
purpose:	*“A directory contains names of files or further directories.”*
restrictions:	$\forall x \in$*directory.elements:*
	$(\exists y \in$*file:* $x = y$*.name*$) \vee (\exists y \in$*directory:* $x = y$*.name*$)$
creation-date:	*date*
protection-mode:	*access-rights*
owner:	*user*
size:	*storage-measurement*

Figure 9. Description of the object *directory*.

The topic of *relations* or *dependencies between attributes* has been already addressed in the context of restrictions which specify relations between values of different attributes. Explicit knowledge about dependencies and relations between attribute values could be used, for example, in planning processes in order to determine side-effects of operations. Since objects affected indirectly by an action must be related to the direct objects of the operation in a specific manner, it should be possible to determine the set of objects affected indirectly using information about the kind of operation and the relationship between the objects.[18]

Another important subject concerned with the representation of objects is the definition of so-called ‘similarity relations’. Objects are considered as *equal* though they are not completely identical if their essential features are identical and objects are conceived as *similar* if they have some essential features in common. They are *identical* if all features are the same. The notion of ‘similarity’ of objects may be useful in plan generation and recognition processes. Consider for example, the sequence of commands *cp fil1 fil2; rm fil1* and the command *mv fil1 fil2*. It is supposed that a plan recognition process regards both actions as being equivalent. But the result of these operations is not completely identical since, in the first case, the creation date of *fil2* is set to the actual date whereas the creation date remains the same if *fil1* is renamed using the *mv* command; there is another, more subtle but relevant difference between the two plans, in case that *fil1* has more than one hard links which will still exist in the first case, i.e. after the *rm* command.

6. Using the Representation of Commands in Question Evaluation and Planning Processes

The main point in the representation of commands is to provide a formal description useful for finding a certain specified command or combination of commands, called *complex command*, and explaining it to the user.[19] A command can be specified in a user's question by describing an action or task she wants to perform or by relating to a certain state of the system she is trying to achieve.

In the first case, a direct mapping takes place from the action mentioned in the user's phrase onto an action defined as a concept in the knowledge base. This mapping is a search process which determines the KB-concept according to

- the action,
- objects of the action,
- attributes of the object, and
- modifiers of the action

contained in the case-frame representation of the input sentence. This process consists of finding a path from the root node of the hierarchy of generic commands to the most specific command which is determined clearly in the user's phrase. Within the current version of SC, the process of determining the KB-concept – the Question Evaluator – uses a decision tree which is based on a classification of commands similar to the taxonomy of generic functions.

When an appropriate command is found, the user has to be informed about the applicability of the command stated in the precondition formula, or a planning process has to be invoked which checks recursively on preconditions of commands selected and thus constructs a plan.

When a desired state – a goal state described in the user's question or a precondition which has to be fulfilled – is to be achieved, the help system has to find an appropriate command causing a respective state change. In this case, the search may again start with a very abstract generic command, e.g. the 'change-object-feature' command. If this is not feasible, the search starts with the respective object and examines the effects of commands operating on it, i.e. those which are connected to it via the attribute 'operations'.

There are some differences between planning in this context and usual planning techniques:

∗ Planning on the Basis of Incomplete World Descriptions

The planner does not operate on complete world descriptions because complete descriptions of the states of a complex system are not feasible. Instead, the planner may use some information about the state of the system with respect to the individual user which is stored in a data base of instances of knowledge base concepts.[20] In addition, the planner should be able to interact with the SINIX system by invoking SINIX commands and thus to obtain requested information about objects, for example, in order to check the truth of preconditions of commands. Another way of extending the data base of facts is to record the results of operations which are known to the help system, for example state changes caused by actions of the help system itself and operations performed by the user which can be simulated by the help system. These simulations of actions in the domain should also take care of relations between objects (see section 4) which may be responsible for side effects of the respective operation.

∗ Planning with Variables

In the case of a help system, a planning process should be able to operate without instantiating variables occurring in schematic operator descriptions since most questions do not relate to concrete objects in the domain but to classes of objects, e.g. *file, user, directory*. Although planning without instantiating variables is not usual, it does not seem to provide fundamental theoretical problems. A possible solution is to introduce "dummy constants" which are substituted for variables occurring in action descriptions. The only difference between this and a "real" plan instantiation is that the truth of preconditions cannot be tested.

∗ Planning with Weighted Preconditions and Effects

As outlined in section 2, preconditions are divided into trivial and severe ones. The planning process should, in a first instance, check only on severe preconditions. Trivial preconditions are not examined until a generated plan is to be executed by the help system itself, or if the help system has to search for the cause of an error message or a problem the user communicated to it. With respect to the effects of a command, an equivalent handling of main and side effects is useful. When searching for a command achieving a certain state of some object, first of all, commands are selected with respect to their main effect. If the desired state is not attainable as a main effect, commands are checked with respect to their side effects.

∗ Planning with an Integrated Hierarchy of Actions and Objects

As already mentioned above, connections between objects and actions represented in the knowledge base allow two ways of searching for a command. The search may start with a generic command and an appropriate command may be found by specifying it successively or, the search starts with an object if a desired state of this object is described.

Finally, a planner for a help system should take into account *knowledge prerequisites* and *postrequisites* (cf. Morgenstern 1987), i.e. information a user must have about the current state of the system – as opposed to 'knowledge preconditions' referred to as 'presupposed concepts' in section 3.2 which relate to concepts a user has to know in order to apply a command. Knowledge pre- and postrequisites are not explicitly modelled in the representation suggested in this paper though the notion of *inform* commands is a first approach to embedding them into reasoning processes.

7. State of Implementation and Recent Developments

The hierarchical structure of the SINIX-KB outlined in section 2 was implemented in Interlisp-D and LOOPS on a Siemens APS 5815 (XEROX 1108-105)[21] and in Franz Lisp and PMFS (Poor Man's Flavor System; an object-oriented offset on Franz Lisp) on Siemens MX-2 and MX500 and DEC VAX computers.[22]

All together, it consists of about 640 concepts, including about 560 actions and 80 objects. The concepts of the action branch are completely worked out but only a small part of the set of objects is described in detail. The semantical representation of commands has not yet been implemented.

In 1988 the Project 'SINIX Consultant' was officially finished and did not receive any more fundings. Nevertheless some students worked on specific aspects and components of the SC system for their Master's Thesis: Bernhard Kipper on the semantics of auxiliaries (Kipper 1991), Manfred Weichel on plan generation (Weichel 1991), Petra Sommer on plan representation (Sommer 1991), Judith Engelkamp on situation semantics (Engelkamp 1991), and Frank Berger on plan recognition (Berger 1990); Frank Berger continued his work in the PLUS-project at the DFKI (German Research Center for Artificial Intelligence) in Saarbrücken (Berger et al. 1993; Thies and Berger 1992; Thies et al. 1992). Following the SC Project, a research group on logic-based plan recognition was founded at the DFKI (Bauer and Paul 1993; Bauer et al. 1993).

Notes

[1] SINIX is a Unix derivative developed by the Siemens AG; Unix is a trademark of X/Open, Inc.

[2] For a survey on the SINIX Consultant system see Kemke (1986), Hecking et al. (1988) and Wahlster et al. (1988).

[3] For the convenience of the reader, German terms designating concepts, attributes etc. defined in the knowledge base are replaced by corresponding English terms throughout this paper.
[4] The object-branch of the current SINIX-KB is not yet developed in detail.
[5] The classification of commands with respect to themes is taken almost directly from the SINIX manual.
[6] Assumptions about the user's knowledge needed by the Answer Formulator in order to generate user-oriented answers are maintained by the User Modelling Module (see Nessen 1987); knowledge preconditions stated as *presupposed concepts* are also used by this module to infer the concepts the user probably knows when using a command correctly.
[7] In this context concepts are regarded as sets of individual objects or, formally, as one-place predicates.
[8] Relevance factors for preconditions have also been used by ABSTRIPS in order to realize hierarchical planning (Sacerdoti 1977).
[9] This approach is related to first order dynamic logic known in the theory of programming languages (Goldblatt 1987) and situational logic proposed by McCarthy and Hayes (McCarthy and Hayes 1969).
[10] In the example outlined, I refer only to files as moved objects, although the same command operates on directories. This could be handled by extending the argument-type to the concept *file* ∪ *directory* (if *file* is not already specified as a subtype of *directory*), or by introducing two different generic functions *move-file* and *move-directory* which are realized by the same command.
[11] The term 'actual-user' is conceived as an *individual concept*, i.e. a concept with (at most) one member (see Brachman and Schmolze 1984). Since it is assumed that exactly one actual user exists, quantification is not necessary. 'Individual concepts' are equivalent to sets with one member and thus may be regarded as individual constants.
[12] Note that dot-notation is used to express the access to attribute values of a concept; these values may again be concepts, and thus, applying the dot-operator iteratively, one first gets, for instance, the *location* of *file*, i.e. *directory*, and then its number of elements by accessing the (*#elements*) slot.
[13] Note that $\mathbf{A}'$ *specializes* **A** is thought of being equivalent to $\forall x : x \in \mathbf{A'} \Rightarrow x \in \mathbf{A}$.
[14] see Schmolze and Lipkis (1983).
[15] For a formal treatment of subsumption of logical formulas see also Kemke (1988).
[16] The second case is identical to so-called 'role-value-maps' in KL-ONE reflecting identity of role fillers or, more general, 'co-referentials' in structural descriptions (Brachman and Schmolze 1984).
[17] Meta-objects are also known in LOOPS.
[18] Constructs for modelling dependencies between attributes are also embedded in several programming languages, for example the concept of 'active values' in LOOPS.
[19] 'Complex command' or 'complex action' is a more general notion than 'plan'. A plan is a (partially) ordered sequence of actions whereas complex actions are actions constructed from simple commands and *combinators*, for example pipes and the sequence operator in UNIX. Combinators are known in mathematics as higher order functions and are used, for example, in the theory of functional programming languages (e.g. 'functional forms' in Backus (1978)).
[20] This corresponds to the A-Box in KL-ONE.
[21] Most of the implementation of the SINIX-KB as well as the development and implementation of the query language KB-Talk were carried out by B. Jung (see Jung 1987).
[22] P. Sommer and M. Weichel converted the SINIX-KB to PMFS-structures and made several improvements and extensions (see Engelkamp et al. 1988).

References

Backus, J. (1978). Can Programming be Liberated from the von Neumann Style? *CAGM* **21**(8): 613–641.

Bauer, M., Biundo, S. & Dengler, D. (1993). PHI A Logic-Based Tool for Intelligent Help Systems. DFKI Research Report RR-92-52, DFKI, Saarbrücken.

Berger, F. (1990). Ein massiv paralleles Modell eines Planerkenners. Diplom-Arbeit, Fachbereich Informatik, Universität des Saarlandes.

Berger, F., Fehrle, T., Kloeckner, K., Schoelles, V., Thies, M. A. & Wahlster, W. (1993). PLUS: Plan-Based User Support – Final Project Report. DFKI Research Report RR-93-15, DFKI, Saarbrücken.

Bauer, M. & Paul, G. (1993). Logic-Based Plan Recognition for Intelligent Help Systems. DFKI Research Report RR-93-43, DFKI, Saarbrücken.

Brachman, R. J. & Schmolze, J. G. (1984). An Overview of the KL-ONE Knowledge Representation System. Fairchild Technical Report No. 655.

Carter, J. (1986). A Taxonomy of User-Oriented Functions. *Int. j. Man-Machine Studies* **24**: 195–292.

Douglas, R. J. & Hegner, S. (1982). An Expert Consultant for the UNIX-Operating System: Bridging the Cap between the User and Command Language Semantics. In *Proc. CSCSI/SCEIO*, 92–96. Saskatoon, Sasketchewan.

Engelkamp, J. (1991). Satzsemantische Repräsentation durch erweiterte Situationsschemata. Diplom-Arbeit, Fachbereich Informatik, Universität des Saarlandes.

Engelkamp, J., Sommer, P., Berger, F., Gintz, R., Kipper, B. & Weichel, M. (1988). Eine SC-Version für UNIX-Rechner. Memo, Fachbereich Informatik, Universität des Saarlandes.

Goldblatt, R. (1987). *Logics of Computation and Time*. CSLI Lecture Notes, Stanford, CA: Stanford University.

Hecking, M., Kemke, C., Nessen, E., Dengler, D., Hector, G. & Gutmann, M. (1988). The SINIX Consultant – A Progress Report. Memo, Fachbereich Informatik, Universität des Saarlandes.

Jung, B. (1987). Wissensrepräsentation in einem intelligenten Hilfesystem. Diplom-Arbeit, Fachbereich Informatik, Universität des Saarlandes.

Kemke, C. (1985). Entwurf eines aktiven, wissensbasierten Hilfesystems für SINIX. *LDV-Forum*, Nr. 2 Dezember 1985: 43–60.

Kemke, C. (1986). The SINIX Consultant – Requirements, Design, and Implementation of an Intelligent Help System for a UNIX Derivative. Memo, Fachbereich Informatik, Universität des Saarlandes.

Kemke, C. (1987). Representation of Domain Knowledge in an Intelligent Help System. In *Proceedings of the Second IFIP Conference on Human-Computer Interaction INTER-ACT'87*, 215–220, Stuttgart, FRG.

Kemke, C. (1988). Darstellung von Aktionen in Vererbungshierarchien. In Hoeppner (ed.), *GWAI-88. Proceedings German Workshop on Artificial Intelligence*, Springer.

Kipper, B. (1991). Semantisch-pragmatische Analyse von Modalverben in natürlichsprachlichen Dialog- und Beratungssystemen. Diplom-Arbeit, Fachbereich Informatik, Universität des Saarlandes.

McCarthy, J. & Hayes, P. J. (1969). Some Philosophical Problems from the Standpoint of Artificial Intelligence. In D. Michie & B. Meltzer (eds.), *Machine Intelligence* **4**: 468–502. Edinburgh: Edinburgh University Press.

Morgenstern, L. (1987). Knowledge Preconditions for Actions and Plans. In *Proceedings of the International Joint Conference on Artificial Intelligence IJCAI-87,* 867–874. Milano, Italy.

Nessen, E. (1987). SC-UM – User Modeling in the SINIX Consultant. Memo No. 18, Fachbereich Informatik, Universität des Saarlandes.

Sacerdoti, E. D. (1977). *A Structure for Plans and Behavior*. Elsevier, New York, Oxford, Amsterdam.

Schmolze, J. G. and Lipkis, T. A. (1983). Classification in the KL-ONE Knowledge Representation System. In *Proceedings of the International Joint Conference on Artificial Intelligence IJCAI-83,* 830–332. Karlsruhe, FRG.

Sommer, P. (1991). SCooPS. Entwurf und Realisierung einer hierarchischen Planrepräsentation für Kommandosprachen. Diplom-Arbeit, Fachbereich Informatik, Universität des Saarlandes.

Thies, M. A. & Berger, F. (1992). Planbasierte graphische Hilfe in objektorientierten Benutzeroberflächen. DFKI Research Report RR-92-13, DFKI, Saarbrücken.

Thies, M. A. & Fehrle, T. & Berger, F. (1992). Intelligent User Support in Graphical User Interfaces: 1. InCome: A System to Navigate through Interactions and Plans 2. Plan-Based Graphical Help in Object-Oriented User Interfaces. DFKI Research Report RR-92-14, DFKI, Saarbrücken.

Wilensky, R., Arens, Y. & Chin, D. (1984). Talking to UNIX in English: An Overview of UC. *Communications of the ACM* **27**(6): 574–593.

Weichel, M. (1991). SCooPS. Ein objekt-orientierter Plangenerator für Kommandosequenzen. Diplom-Arbeit, Fachbereich Informatik, Universität des Saarlandes.

Wahlster, W., Hecking, M. & Kemke, C. (1988). SC – Ein intelligentes Hilfesystem für SINIX. In Gollan, Paul & Schmitt (eds.), *Innovative Informationsinfrastrukturen*. Springer.

Wilensky, R. (1984). Talking to UNIX in English: An Overview of an On-line UNIX-Consultant. *AI Magazine* **5**(1): 29–39.