EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Remarks on a few concepts in artificial intelligence . KNOWBODE : a case study in knowledge engineering

van Liempd, Gidi

Award date:
1985

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

PART 2
KNOWBODE: a case study in knowledge engineering

Gidi van Liempd

Afstudeerverslag

Afstudeerhoogleraren: prof. ir. O. Rademaker
prof. ir. F.J. Kylstra

Afdeling der Technische Natuurkunde TH Eindhoven

Contents

p. 1 Introduction

p. 4 Chapter 1: What does SAGE do?
p. 5 §1. The SAGE expert systems package
p. 5 §2. The SAGE modelling language
p.11 §3. The SAGE executive
p.12 §4 The SAGE interactive command language
p. 14 Chapter 2: KNOWBODE: an expert system for making Bode plots
p. 14 §1. The task
p.14 §2. The expert knowledge
p.16 §3. The expert system
p.17 Chapter 3: A few dialogues with KNOWBODE
p.18 §1. Dialogue 1
p.20 §2. Dialogue 2
p.26 §3. Dialogue 3
p.32 §4. Dialogue 4
p.35 Chapter 4: Analysis of KNOWBODE
p.35 §1. The structure of dialogues in the SAGE system
p.36 §2. The model
p.40 §3. The interface and external environment
p.42 §4. Speed and size
p.44 Chapter 5: The representation problem
p.46 §1. Design Criteria
p.46 §2. Performance
p.46 §3. Understandability
p.47 §4. Flexibility
p.49 Chapter 6: How to solve a problem using the SAGE building blocks
p.50 §1. How to make a Bode plot
p.51 §2. Interacting subproblems
p.53 §3. How to translate the flowchart into the SAGE building blocks
p.54 §4. Brief summary
p.55 Chapter 7: AREAs and ACTIONs
p.56 §1. Example: initialization of bac=kground memory
p.59 §2. ACTIONs which should not appear in the flowchart
p.62 Chapter 8: How to make a goal tree in SAGE
p.62 §1. Investigation will lead to evaluation
p.65 §2. Evaluated facts will not be reinvestigated
p.66 §3. The order of applicable RULEs
p.68 §4. How to make an understandable tree
p.72 Chapter 9: Design of the Interface and the external environment
p.73 §1. Historical account of the design
p.78 §2. Grey zone
p.80 Chapter 10: Multiple use of knowledge
p.80 §1. In what kind of situations do we encounter sequences of similar operations?
p.81 §2. What possibilities are there in SAGE for loops and subroutines?
p.83 §3. Then how should we handle situations with loops?

Introduction

This report is the result of research done by Gidi van Liempd, as part of a research project on the use of AI (Artificial Intelligence) in control engineering. This project is under the supervision of the group S&RT of the department of Physical Engineering, Eindhoven University of Technology.

The report is divided into three parts, of which this is the second.
In the first part, remarks on a few concepts in Artificial Intelligence are made.
This part contains a description of a program called "KNOWBODE" made by the author, on the expert system generator SAGE (by SPL). The purpose of that part of the report is to gain insight on how SAGE works, which may be useful for designers of expert systems (in SAGE), the so-called "knowledge engineers".
The third part only contains the source listings of the KNOWBODE program and associated programs necessary to run it, if desired, on any SAGE system.

This part of the report was written with one type of reader primarily in mind: a knowledge engineer (or actually, someone who wants to become one). What is a "knowledge engineer"? According to Feigenbaum, quoted in [17] (p. 140):

"The knowledge engineer practices the art of bringing the principles and tools of AI research to bear on difficult applications problems requiring experts' knowledge for their solution. The technical issues of acquiring this knowledge, representing it, and using it appropriately to construct and explain lines-of-reasoning, are important problems in the design of knowledge-based systems....The art of constructing intelligent agents is both part of and an extension of the programming art. It is the art of building complex computer programs that represent and reason with knowledge of the world."
An "expert system" is such a complex computer program which contains expert knowledge. Or, in the words of the SAGE manuals: ([20]SAG01, p.6:)
"An expert system is a computer program which reflects the decision-making process of a human specialist. It embodies organised knowledge containing a defined area of expertise, and frequently operates as a skilful, cost-effective consultant." SAGE is an "empty" expert system, in which knowledge for various tasks may be encoded.
However, since the author is aiming at knowledge engineers, he thinks it is better to stress the fact that expert systems are <u>programs</u>, not just (static) bodies of knowledge, but structured sequences of actions with a goal (like making a Bode plot). And it are the knowledge engineers who must make such programs!

In the first chapter I shall describe the components of SAGE, i.e. the building blocks of the structure which embodies the program. This whole report is an attempt to show the reader how those basic components may be put together to create the effects desired in some program (written in

the SAGE modelling language).
These effects have three aspects (which are briefly discussed in Chapter 5): performance (a construction should do something), understandability (the user or designer must be able to understand the construction) and flexibility (constructions should match with other constructions, and preferably be general enough to be used in other sections of the program as well). Various aspects of this problem are discussed in Chapters 6 through 10.
Most of these aspects are illustrated by examples from the program KNOWBODE, an expert system in SAGE which is able to describe some features of Bode plots (a graphical means of expressing the frequency-transfer function of a system). In Chapter 2 the necessary expert knowledge is briefly described, and in Chapter 3 several dialogues with the program are displayed, illuminating some of the possibilities of the SAGE system. Chapter 4 contains an analysis of a few quantitative aspects of the KNOWBODE program, and Chapter 10 some conclusions.

I shall not refrain from attributing SAGE humanoid qualities, because I think they give an easy and useful description of the behavior of the program, which is also easily remembered. Naturally, such a description will give rise to errors for those working with it, since SAGE is not exactly humanoid, but such an error may possibly be treated as another trait of character, and does not outweigh the advantages of the method.

Sometimes concepts or guidelines for working with SAGE could be better explained by using examples from another program by the author, called MOVER.
This program, which operates in a "blocks world" and is able to describe how it would put one block on top of another one (if it was connected to a robot), is based on a procedure mentioned in [19] p.35-41.
Since the examples used are intended mostly to convey a general flavor of working in SAGE, and do not rely heavily on some (possibly suggested) performance, it was not deemed necessary to include the source listing of this program in part 3 of this report.

Probably not enough stress has been put in this report on the fact that all examples from SAGE can be made visible during interactive sessions. The format chosen to display the examples in this report is usually the format in the source listings. This is done for a more uniform approach of the texts taken from KNOWBODE. See Chapter 3 on how it looks in an interactive session.

It may be clear that this report may be regarded as an extension of the SAGE manuals. There, it was discussed what the basic components do. Here, how to make constructions with them, to realize (pieces of) program for some (high-level) task.
As most manuals, it will probably be very boring to read. Any sensible reader should skip the parts he probably knows (readers familiar with SAGE may skip Chapter 1, readers familiar with control engineering the first two sections in Chapter 2).

I'd like to thank Hans Canters, for being a very friendly representative of SPL.

Chapter 1
What does SAGE do?

In this report we are investigating SAGE, an example of a working expert system.
What is an "expert system"? To quote from [1] the (slightly modified) description given by the British Computer Society's Committee of the Specialist Group on Expert Systems:

> "An "Expert System" is regarded as the embodiment within a computer of a knowledge-based component from an expert skill in such a form that the machine can offer intelligent advice or take an intelligent decision about a processing function. A desirable additional characteristic, which many would regard as fundamental, is the capability of the system on demand to justify its own line of reasoning in a manner directly intelligible to an anticipated enquirer. The style adopted to attain these characteristics is rule-based programming*."

I shall not discuss here the implications of this description, but do point out that because of the rule-based programming and the system's explanation capabilities, the advice given or the decision taken is regarded as "intelligent" rather than (just) the output of a complex program. (See part 1 of this report).

On working with an expert system, we are considering the following situation:

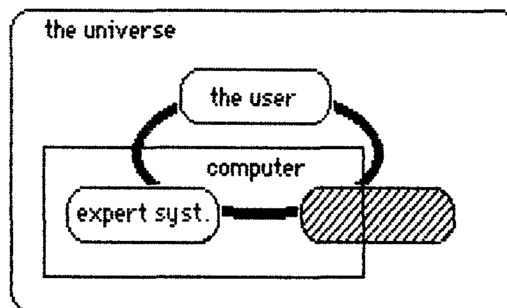


fig. 1 global situation

The shaded box represents a part of the universe, in which the user is interested, and on which he and/or the expert system can act.
Physical instances of this shaded box could be almost any system: ranging from a "blocks world" + a robot-arm and eye + computer programs to control that arm and eye, to just another computer program (maybe even a copy of the expert system itself). A part common to all such systems, can be a system supplementing the possibilities of the expert system (e.g. in the case of SAGE it may provide a "long-term memory" function by storing results in files).

*** By which is meant that the expert knowledge is encoded in rules essentially based on the structure "IF condition THEN action".**

We will call this part of the shaded box the "External Environment" with respect to the expert system.
This report, which discusses the expert system SAGE, may therefore also contain a discussion on suitable forms of the environment external to SAGE (Chapter 9).

§1. The SAGE expert systems package

SAGE is an expert systems package, which has two major components: the Compiler and the Executive. It also provides the SAGE modelling language, in which models (='programs') may be written, and the SAGE command language, for user interaction with the Executive.
The solution of some (expert) task may be encoded in a MODEL using the SAGE modelling language. The compiler must then be used to translate the source form of this program (off-line) into a representation suitable for the executive. The executive controls the interaction between a user and a model, hence controls execution of the model, including the interaction of the model with its external environment.
Since the model can represent all kinds of expert tasks, SAGE is often referred to as an "expert system generator" rather than an "expert system". It's like EMYCIN (or empty MYCIN), which can be employed with various information for different applications, but which derived from MYCIN, an expert system which advises on the diagnosis and treatment of bacterial infections (see [1], § 3.4).

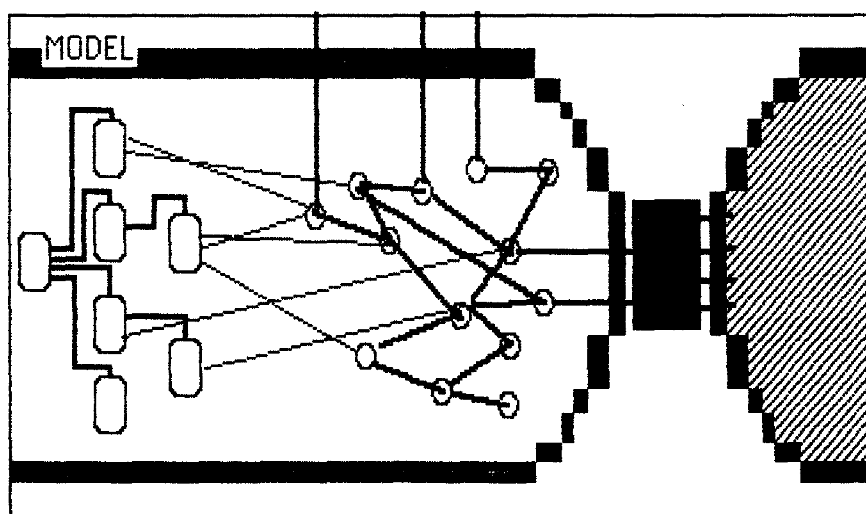


fig. 2 a SAGE MODEL

§2. The SAGE modelling language

One way of describing a SAGE model is to say that it consists of a hierarchy of AREAs, which is able to do something because it governs a forest of (goal)trees. The basic components of AREAs are called "ACTIONS", those of the goal trees "RULES". We shall now descibe the two parts:

1. ACTION-based part,
2. RULE-based or goal tree part.

§2.1. The ACTION-based part

The basic building blocks of the ACTION-based part are called "AREA" and "ACTION".
As we shall see later, AREAs may embody functional entities, chunks of program to achieve a high-level goal, like "getting the FTF". An executable SAGE model consists of a non-empty AREA designated "FIRSTAREA" (which is where the program normally would start running), and (optionally) several other AREAs.
An area is introduced by a declaration:

```
AREA areaname : string
```

where "areaname" identifies this area within the model, and the string should describe the purpose of this area. (Such kind of strings are attached to most SAGE building blocks.)
The ACTIONs (and the assertions, objects, rules and questions which we shall describe in the RULE-based part) following this area declaration and preceding the next area declaration are said to belong to this area.
When an area is being investigated, the actions belonging to it are processed in the order of their declaration in the model source.

An ACTION is a basic program unit. It can in some ways be compared to a sequence of a few statements in a programming language like FORTRAN.
Each action, like an area, has an identifying name and defining (and optional explanatory) text. It has an optional precondition whether or not to execute the actionbody. This body of the action declaration specifies the effects the action is to have; actions may roughly be divided into two (interdependent) classes:

1. those concerning the explicit flow of control
2. those concerning program output.

<u>§2.1.1. action effects concerning the flow of control:</u>

```
STOP
RESTART
REASK questionlist
CONSIDER goallist
FORGET goallist
```

are action effects with the primary aim to control explicitly the flow of the program. We will elaborate on them in Chapter 7 and 9, but for the moment only illustrate the general idea by an example:

```
ACTION Bode_plot :
"I will now calculate a Bode plot"
        CONSIDER user_wishes
        ALSO
        CONSIDER get_FTF, prep_low_freq\break,
                xmine_low_freq\break, xmine_intervals
        PROVIDED Bode_plot_wanted
```

The action effects in <Bodeplot> will (all) be executed when the precondition, the statement <Bodeplot_wanted> (associated string: "the user wants me to calculate a Bode plot") has been evaluated and found to be true.
An action effect like "CONSIDER user_wishes" will transfer control to the AREA user_wishes, a chunk of program for determining more precisely what kind of results the user wants. The ACTIONs in that area will then be processed (in the course of which again other AREAs may be entered), and afterwards control is returned to <Bodeplot> where the next action effect ("CONSIDER get_FTF") will be executed.
As you can see, "CONSIDER area" is in some ways equivalent to calling a subroutine in a FORTRAN statement.
But "CONSIDER" can also be used to start investigation of an assertion or object, which in SAGE means trying to get a (truth or certainty) value for some statement. This means investigating a goal tree, where the root or top node is formed by the assertion/object to be CONSIDERed. This is done in the RULE-based part of the model, which we will consider in a moment.

§2.1.2. actioneffects concerning output of the program:

> ADVISE [LATER] advisebody (,advisebody)
> CALL [LATER] procedurename parameterlist

are action effects which take care of output of the program.
An ADVISE action is analogous to a print statement in conventional programming languages. ADVISE is followed by a list of strings, assertion names, object names or constant names. When the action takes effect the strings and the values of the specified assertions, objects or constants are displayed in the order specified.
Assertions or objects in the advise list which have not yet been investigated are all investigated, in advise order, before any of the advice is given. This means that ADVISE, like CONSIDER, can also start the evaluation of facts (i.e. ASSERTIONs or OBJECTs).
SCALES can be attached to assertions/objects to convert numerical values into strings, thus allowing a more understandable presentation of results.
An example:
(an action effect from ACTION <advice_about_|H|>)

```
ADVISE
"!N", behavior_|H|, "represents a ", kind_of_line
```

can produce as output (depending on the actual values of <behavior_|H|> and <kind_of_line>):

> For all frequencies, log|H| vs log(w) represents a horizontal line.

Here the actual values have been converted using SCALES.

User-defined PROCEDUREs and FUNCTIONs (see Chapter 9) are the means by which the model may communicate with the outside world other than through the terminal (sensors, actuators, files, databases etc.)
FUNCTIONs may act upon this external environment and retrieve values from it. They are part of the leaves of the goal trees.
PROCEDUREs only act upon the outside world, and are executed by using the CALL action effect. The parameters to be used as input for the procedure will be evaluated upon CALL. This means again that "CALL", like "CONSIDER", implicitly can start the evaluation of facts.
As an example, examine

```
ACTION add_break_point :
"finding the break point of a factor (if possible) "
"and adding it with the factor to a list"
        CALL add(break_point, cause_factor, list)
        PROVIDED break_point_exists
```

This action will do as the associated text says. The list is stored in the external environment, so a procedure (<add>) must be used to put the values there. Notice that it works without the programmer explicitly having had to mention something like "now you must investigate what value <breakpoint> has". It is as if <breakpoint> is a call to a subroutine to compute that value, but actually this is done in a goal tree.

§2.2. The RULE-based or goal tree part

So on the one hand we have AREAS and ACTIONs, to be regarded as subroutines and statements respectively, and on the other we have goal trees. What are those goal trees made of?

The main component in the reasoning structure in a SAGE model is the ASSERTION. An ASSERTION should be associated with a statement whose truth is to be determined in a certain situation. An example of an ASSERTION declaration is:

```
ASSERTION read_FTF :
"I read the FTF which you gave me"
        DEFAULT FALSE
```

("I" is meant to be SAGE, "you" the user).
In the SAGE structure the likelihood of an assertion must be expressed by a real number associated with that assertion. This real number always lies in the range -100.0 to 100.0, the latter representing TRUE and -100.0 representing FALSE. Numbers with absolute values less than 100.0 represent degrees of belief which are not absolutely certain. Zero represents complete uncertainty. Apart from that, an assertion may also have an "unknown" likelihood, if all available methods of establishing it fail.
The degree of belief of an assertion is connected to the probability that the assertion is true via a logarithmic scaling function (see [22]SAG03, p.59).

Before we discuss the means to establish the value of an assertion, let us first consider another component of SAGE, quite similar to the ASSERTION. SAGE allows the use of entities which have associated with them real numbers which can lie in any range. These entities are called OBJECTs. They are declared in a similar way as assertions, but must contain a specification of the range within which their numerical value lies. For instance:

```
OBJECT break_point :
"the break frequency"
MORE
"the frequency where one region of asymptotic "
"behavior of gain |H| and phase arg(H) meets "
"another"
            (0.0, maximum_real)
```

As mentioned, the ASSERTIONs and OBJECTs can get a value depending on the actual situation. Let "fact" mean either assertion or object, then investigating a fact means the program tries to establish a value for it. How does the program go about doing that?

RULEs are the means to establish the value of a fact by using the values of other facts, or by using FUNCTIONs to get a value from the external environment.
For instance:

```
RULE calc_break_point_2 :
"below the break point frequency, a type 3 (= "
"2nd order) system has a horizontal asymptote "
"in |H| and tends towards 0 in arg(H), above it "
"arg(H) tends towards + or - 180 degrees, and "
"log|H| vs log w has an asymptote sloping with "
"tangent + or - 2. The break frequency is "
"multiplied by 1/T if there is a time constant T "
"in this type 3 system"
            break_point IS square_root(alpha*alpha +
                                       beta*beta)
            PROVIDED type(factor) = second_order
```

This rule can be used to establish a value for <break_point>. There can be several different rules for different situations.
The facts in the precondition (only <factor>, second_order is a CONSTANT) and the rule body (<alpha> and <beta>) have values which also depend on rules and questions. So if the program investigates <break_point> by using rule <calc_break_point_2> it will have to suspend that investigation to examine the values of the facts in the precondition first. When the precondition expression has been evaluated to a value between 0.0 and 100.0 (TRUE), the facts <alpha> and <beta> must be investigated.
Similarly, a DEFAULT for a fact can be the name of another fact which needs investigating.

The procedure for investigating a fact is as follows:

a) If it is marked ASKABLE (in the source text) it is asked; if the user does not reply "UNKNOWN", its value is determined from the user reply and the investigation of that fact is completed.

b) If the fact is askable but the user answers "UNKNOWN", or not ASKABLE, then each QUESTION and each RULE which gives a value to the fact is tried, in the order in which they occur in the model source text, until one succeeds.

For a QUESTION to succeed, the question preconditions must be met, and the user must not reply "UNKNOWN".

For a RULE to succeed, the rule preconditions must be met, and the rule body must not evaluate to "UNKNOWN".

c) If all questions and rules defining the value of a fact fail, the value taken for the fact is the DEFAULT value if one is specified, otherwise the value is deemed "UNKNOWN".

Once the program has found a value for a fact, it will not (easily) forget or change that value. This is called "truth maintenance". A direct effect is that the program won't ask any question more than once.

The whole structure of facts, associated with rules to link them to other facts, which in turn are linked to yet other facts (or to the outside world via questions or functions), can be compared to a <u>tree</u> (Winston,[19] p.33).
The tree has <u>nodes</u>, corresponding to ASSERTIONs and OBJECTs. The top node, the one being investigated in the course of some ACTION, may also be considered the <u>root node</u>, in SAGE called the <u>goal</u>*.
Nodes are connected to other nodes by <u>branches</u>, corresponding to RULEs.
The <u>leaves</u>, the points where the tree receives its light and carbon-dioxide, correspond to the points where the facts get values from the outside world: the ends of QUESTIONs, the ends of RULES (CONSTANT values, FUNCTION values).
Investigating a yet unevaluated node means, in this analogy, finding paths to the leaves which will provide the node with sufficient oxygen.
And since the investigation begins at the goal and works its way backwards towards the things which may already be known, it is called <u>backwards chaining</u> (Hofstadter [4], p.618).

*** Notice that 1) this top node can be an intermediate node in another tree!, and 2) the user can make <u>any</u> node a goal in an interactive session using the command "CONSIDER". By the way, it is not true that the top node is called "goal" in general in SAGE: only when it is a precondition of an ACTION or explicitly CONSIDERed. I prefer to use the word "goal" whenever a fact appears in an ACTION, even as a parameter in a function call.**

§3. The SAGE Executive

"Running a SAGE model" must in my opinion be regarded as running a program in a special environment. This environment is called the "SAGE executive", and it is special because it allows a richer interaction than a "normal" program environment would do.

The SAGE executive controls the interaction between a user (a person at a terminal, or other software systems) and a model (program). The user can conduct his part of the interaction by means of the SAGE interactive command language. The design of the model must be such, that the interaction of the model and the user is as good as possible in terms of the design criteria (performance, understandability and flexibility: see Chapter 5).

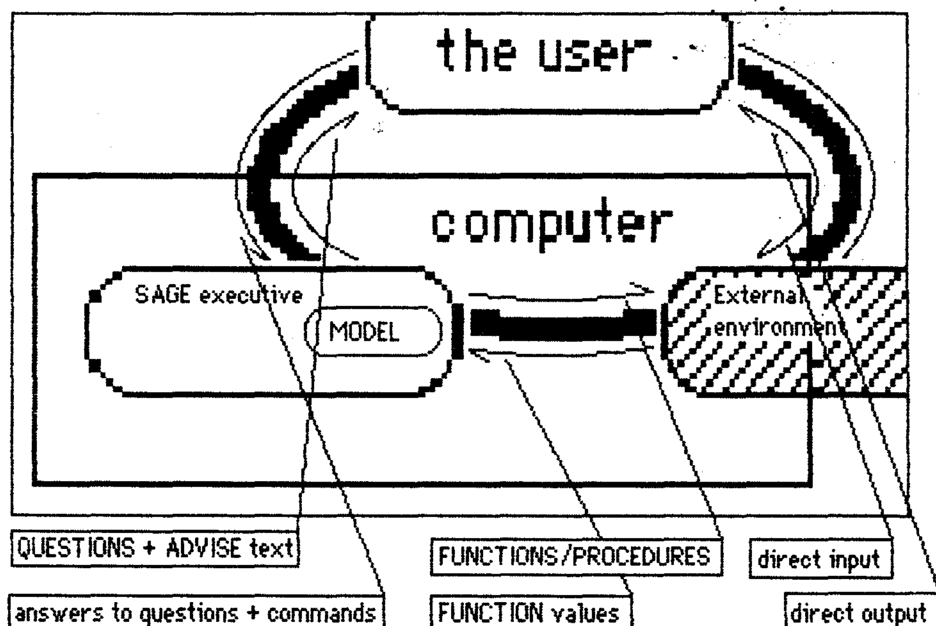


fig. 3 Interaction between parts

In the SAGE terminology, the executive can be in one of four states or "modes":

1. pre-model mode
2. questioning mode
3. (user) command mode
4. done mode.

Pre-model mode is the initial state of the executive before a model is loaded.
Questioning mode is the state during a consultation with a model, when the executive is evaluating the model, asking the model's questions and giving model-defined output. This is the "program-running" mode.
In this state the user can ask for some explanation of the system, e.g. ask for a model question to be explained ("?" command) or ask for the reason (context) why the system asked a particular question ("WHY" command).
Further explanation can be obtained when the user interrupts the program, i.e. changes the state to command mode. Then, the user can ask for more types of information, but also influence the sequencing of the consultation

session. He can for instance leave certain parts of the model out of the run ("FORGET"), or concentrate the investigation on one or more particular goals.
Running the program can then be resumed, until the user interrupts it again or the evaluation of goals is finished. The latter is called the "done mode", which again allows for various kinds of explanation (e.g. "FACTORS", "LIST DONE GOALS" etc.).

§4. The SAGE interactive command language

The SAGE interactive command language consists of several commands the user can type in on the terminal, in order to interact effectively with a model.
A more detailed description is included in Appendix B, [21]SAG02 and [22]SAG03.
For the moment it suffices to list the commands in classes, which I will divide in two groups:

1. Sequence commands
2. Explanation commands.

§4.1. Sequence commands

Sequence commands are all commands which will directly or indirectly influence the course the program takes.
They can be subdivided into 5 classes:

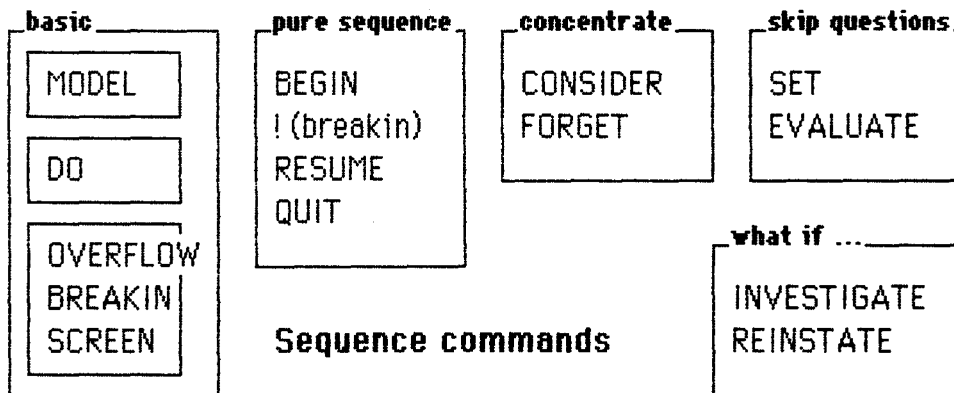


fig. 4 Sequence commands

More information about each command can be found in Appendix B. Examples of the use of these commands can be found in the dialogues in Chapter 3.

§4.2. Explanation commands

Explanation commands are all commands which may help the user understand (parts of) the program and its execution, and the commands to record consultation sessions.
They can be subdivided into 4 classes (see fig. on next page):

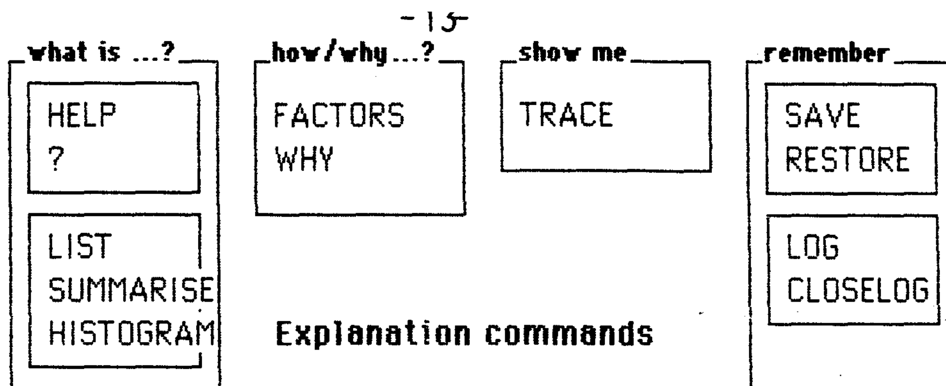


fig. 5 Explanation commands

Explanation commands may be used both by the user and the designer, i.e. to test a certain consultation session (a run of the program), or to test the model used in that session. More information about these commands can also be found in Appendix B, and the dialogues will provide many examples on their use.

Chapter 2

KNOWBODE : an expert system for making Bode plots.

§1. The task

Since my work on the SAGE expert system represents part of a research project on the use of AI in control engineering, it was decided that an example expert task to be implemented should be selected from that field. A task that according to many students requires considerable expert knowledge in control engineering is passing a test in "Dynamische Systemen" (i.e. "Dynamic Systems", an elementary course in control engineering)
The expected size of this task* plus the fact that a primary aim of the research is the investigation of the possibilities of the expert system, made me choose a much smaller task: how to make a Bode plot.
I expected SAGE to be good at purely consultational tasks (evaluating statements in terms of other statements) possibly supported by some numerical procedures in the external environment, but I decided to test its limits. (Simon,[15] p.13:) In a taxing [task] environment we would learn something about its internal structure, specifically about those aspects of the internal structure that are chiefly instrumental in limiting performance.
In making a Bode plot calculations are essential: if SAGE was to be a consultant on this, the "numerical" part (as opposed to the "symbolical" part) could not be placed external to the expert system, but should be a central part in it. Will this create any special problems (and why), in addition to the central problem: how to make an expert system in SAGE?

§2. The expert knowledge

The domain specific knowledge needed to compute a Bode plot can be found in "Dynamische Systemen" [12] p., but also in books like [3],§ 6-4.2. From this book I will reproduce a small part giving an impression what a Bode plot is, for those not familiar with the term. Notice that this text, from which the knowledge needed in the system can be taken, could also be part of the introductory text of the expert system.

In control engineering, the frequency-transfer function (FTF) of a system is very useful as a means of predicting (control) system response and specifying performance ([3], p.140).
The FTF of a system is a complex frequency function T(jw) which tells us that a stable linear time-invariant system subject to a harmonic input signal of frequency w will, in steady state, have a harmonic output signal of the same frequency as the input but with the amplitude magnified in the ratio |T(jw)| and with the phase advanced with the amount $\angle T(jw)$. ([3], p. 140).

* The textbook "Dynamische Systemen" ([12]) contains 250 pages. If we roughly equate the knowledge contained in one sentence with the knowledge in one RULE, several thousand rules would be necessary. In chapter 4("analysis") we will discuss the size of the KNOWBODE system.

Bode suggested a practical plotting method of the FTF, which is particularly suited to those cases where the FTF appears in the form

$$T(jw) = K \frac{(1 + jw T_1)(1 + jw T_2)\cdots}{(jw)^N(1 + jw T_a)(1 + jw T_b)\cdots}$$

By taking the logarithm of this equation, and comparing the real and imaginary parts of it, we obtain the following two expressions for the *gain* of FTF, expressed in db, and the *phase*, expressed in radians :

Gain:

$$20 \log |T| = 20 \log K + 10 \log[1 + (wT_1)^2] + \ldots$$
$$- 20N \log w - 10 \log[1 + (wT_a)^2] - \ldots$$

Phase:

$$\angle T = \tan^{-1}(wT_1) + \ldots - N\pi/2 - \tan^{-1}(wT_a) - \ldots$$

It is particularly simple to plot these functions on a semilog plotting paper (see fig. 6). On such a paper, the function $-20N \log w$ will simply be a straight line with a negative slope of $-20N \log 2 \approx -6N$ db per octave.

The terms of the form $\pm 10 \log[1 + (wT)^2]$ are also handled very conveniently. We note that if $wT \gg 1$, that is, $w \gg 1/T$, then this term approaches the asymptote $\pm 20 \log(wT)$. This is obviously a straight line with the slope ±6 db per octave intersecting the 0-db level at the frequency $w = 1/T$, referred to as the *break frequency*.

If $wT \ll 1$, that is , $w \ll 1/T$, then the term approaches the asymptote $\pm 10 \log 1$, which is obviously the abscissa axis. Right at the break frequency, the term equals $\pm 10 \log 2 \approx \pm 3$ db. The total plot is shown in fig. 6. To obtain the overall gain, one only has to add up terms which consist basically of straight-line segments. The phase angles of these terms, $\pm\tan^{-1}(wT)$, are also plotted in the graph. *It is important to note that the total gain and the total phase of T(jw) are obtained simply by summing up terms of the basic types shown in fig.6.*

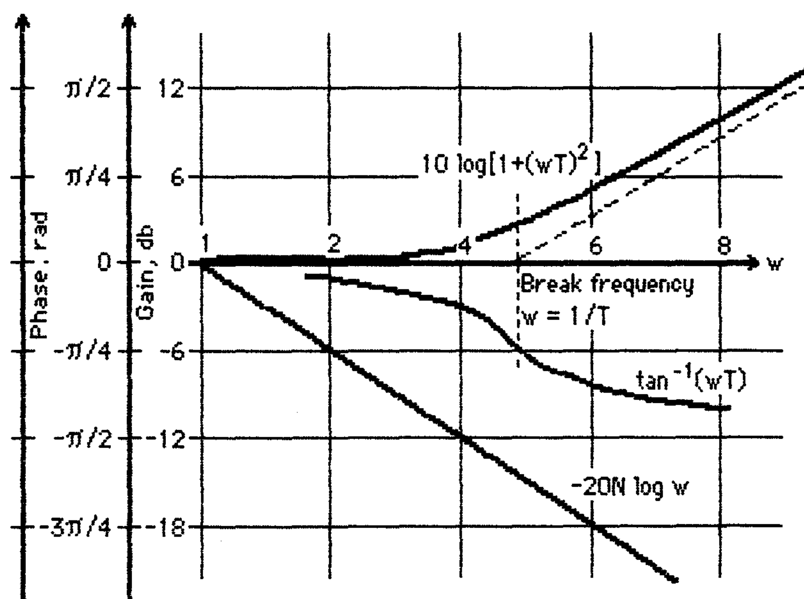


fig. 6. example of a Bode plot

S3. The expert system

KNOWBODE is a computer program describing some features of the Bode plot corresponding to a frequency-transfer function consisting of several (up to 10) factors.
Description of the plot is necessary, because only an "expert" on Bode plots needs the mere picture to see all features. Non-experts need to be told what they see.
By the way, KNOWBODE does not draw the actual Bode plot, because we are mainly working on a (simple) video terminal, and SAGE itself has limited graphical abilities*. But it is possible to add an external FUNCTION to the program for drawing the plot on some plotter or video screen.
KNOWBODE can handle only a small portion of the complete task "making a Bode plot". It can interpret only frequency-transfer functions provided in factorised form. It has limited capacities for interpreting symbolic variables (like "tau-1"). It will describe features concerning the gain |T|, but not yet those concerning the phase /T. It does not describe (or detect) extremum points in the gain.
Some of these limitations may probably be overcome by adding more ACTIONS/RULES etc. Others are more fundamental, due to limitations in SAGE and the representation chosen for the problem. In the following chapters, describing the design of KNOWBODE, I shall also try to indicate the nature of some of these constraints.

* In our case study "the introduction of variables" (Appendix D) we will briefly discuss a possibility for graphical output.

Chapter 3
A few dialogues with KNOWBODE

This rather long chapter may give you some idea of the possibilities of working with SAGE.

In the first dialogue, we show a run of the program KNOWBODE which is deliberately kept as simple as possible. It may give you an idea of the questions which are usually asked by this program, and show the usual output.

In the second dialogue, we shall deal with a few general features. The conversation shows that the user can ask "why", "what is..","what if.." and related questions, and inspect parts of the program.

In the third dialogue, we are witnesses of the execution of the program on a few levels, and use the question possibilities the user has to get some task-specific information.

In the last dialogue, the user takes over control of the program, and meets a few limits.

All dialogues are done under the assumption that the user is rather familiar with the SAGE system, but not with KNOWBODE. Other users however will probably have to make frequent use of the HELP option. Readers of this report can use Chapter 1, Appendix A and B and the HELP files included in part 3.
To distinguish the user's answers and commands from the system output, we asked the user to type in capitals.
Everything in the dialogues in a font like this should be regarded as the remarks of an observer, who has a rather detailed picture of both KNOWBODE and the user's mind. Remarks between string quotes "like this" should be regarded as an approximation of the user's thoughts on some decision. N.B. this observer is not part of SAGE, nor is it a computer program.
Superscript numbers, like this[1], used in the first dialogue, are added in this report to mark certain points. Between those points, the response time of SAGE was measured: see fig. 11 on p. 42.

S1. Dialogue 1

(In which a user familiar with SAGE runs KNOWBODE just to get an answer. It may give you an idea of the normal run of the program.)

>RUN SAGEE

SAGE Expert System Executive, version 1.3
Copyright (C) SPL International 1981.
Licensed Site: University of Eindhoven, on PDP 11/34

If you do not know what to do at any time, type HELP

Please type a command : MODEL KNOWBODE

Current model is <Dynamic_Systems> - Version 2. 28-September-84.
Author: Gidi P. van Liempd
Compiled at 16:51:00 19-Dec-84

Type BEGIN to begin.

Please type a command : BEGIN(1)
(Now the program to compute a Bode plot will start running)

(2)In this version of <Dynamic_Systems> the only thing I can do is calculate a Bode plot.
-- more (Y/N) ? NO(3)
(the user doesn't want the extra introductorytext)

(4)Do you want me to calculate a Bode plot?
(You may answer yes or no) : YES(5)

(6)Are you familiar with me?
(Range is -5 -> 5) : YES(7)
(however, if <u>you</u> (the reader) are not familiar, see the next dialogue)

(8)When I give you the Bode plot, do you wish me to explain in more detail the contribution of each factor?
(You may answer yes or no) : NO(9)

(10)At what points during the process of calculating the Bode plot do you wish to have the opportunity to interrupt me :

1. at all possible points
2. I don't care, just give me the Bode plot
3. I want to have the opportunity to interrupt you at the point where you examine individual factors for their interesting frequencies
4. I want to have the opportunity to interrupt you when there is more than one factor causing the same break point
5. I want to have the opportunity to interrupt you when you are about

to examine a new break frequency

Make a choice : 2[11]

[12]Is there anything you want to ask?
(You may answer yes or no) : NO[13]

[14]Is the frequency-transfer function available?
(You may answer yes or no) : YES[15]

[16]You may now type in the frequency-transfer function (FTF):
1/(1 + jw 3)?[17]
(A simple FTF. The '?' is necessary to mark the end)

[18]For low frequencies, the asymptote of |H| in a log|H| vs log(w) plot represents [19]a horizontal line. For decreasing frequency, log|H| will approach the [20]asymptote from below.
Above w = 0.33 the behavior of the asymptotes of |H| and arg(H) changes due to [21]factor FACTOR-1. The previous asymptote intersected the break point frequency at |H| = 1.00, but the actual value of |H| at the break point is :
0.71.

The effect from this break point on the total asymptote of |H| is, that the [22]slope changes to -1.00.

*** That concludes the consultation. ***

Type QUIT to quit.

Please type a command : QUIT
>

S2. Dialogue 2

(In which an inquisitive user, who is familiar with SAGE, is having a go at KNOWBODE. It will demonstrate a few of the possibilites in interactive sessions.)

>RUN SAGEE

SAGE Expert System Executive, version 1.3
Copyright (C) SPL International 1981.
Licensed Site: University of Eindhoven, on PDP 11/34

If you do not know what to do at any time, type HELP

Please type a command : MODEL KNOWBODE

Current model is <Dynamic_Systems> - Version 2. 28-September-84.
Author: Gidi P. van Liempd
Compiled at 16:51:00 19-Dec-84

Type BEGIN to begin.

Please type a command : BEGIN

In this version of <Dynamic_Systems> the only thing I can do is calculate a Bode plot.
-- more (Y/N) ? YES
(the user: "Yes, let me see some more text on this")

This Bode plot is calculated using a frequency-transfer function, which in this version I can only use if it is given in factorised form. I mean, if the FTF is not available, but e.g. a differential equation on the system, there is not much I can do in this version.

Do you want me to calculate a Bode plot?
(You may answer yes or no) : WHY
("Why? What are the consequences of my answer?")

Your answer supplies a value for the assertion <Bode_plot_wanted>
the user wants me to calculate a Bode plot

Please reply : WHY LEVELS 3
("Your explanation was not enough. Please give me a few more levels of explanation on the consequences")

Your answer supplies a value for the assertion <Bode_plot_wanted>
the user wants me to calculate a Bode plot

The assertion <Bode_plot_wanted> is a precondition of the action <Bode_plot>

I will now calculate a Bode plot
in consultation area <control>
the overall sequencing

Please reply : LIST BODE_PLOT
("Well then, what does this action <Bode_plot> do?")

Action <Bode_plot>:
I will now calculate a Bode plot
Effects are:
CONSIDER what advice the user expects from me <user_wishes>
CONSIDER how to obtain the frequency-transfer function (FTF) <get_FTF>
CONSIDER initialization of increments and background memory space necessary for area <xmine_low_freq\break> <prep_low_freq\break>
CONSIDER investigation of the low frequency behavior of the gain |H| and phase arg(H) for the FTF H, and preparation of a list of frequencies where a change in the behavior of |H| or arg(H) will occur <xmine_low_freq\break>
CONSIDER the initialization of increments in background memory necessary for AREA <body_xmine_intervals> and start of the actual calculation <examine_intervals>
Provided the given precondition is fulfilled.

Please reply : ??
("Could you please repeat the question for me?")

Do you want me to calculate a Bode plot?
(You may answer yes or no) : YES
("Okay, I'll give the program a chance")

Are you familiar with me?
(Range is -5 -> 5) : ?
("<u>What</u> do you mean by that?" By the way, do you see that there is a <u>certainty range</u> (instead of "YES or NO") connected to this question!)

I mean, have you worked with me (SAGE) before, do you know some of the commands available to you etc.
Please reply : YES
(This is the same as answering "5", i.e. "I am certain on this")

When I give you the Bode plot, do you wish me to explain in more detail the contribution of each factor?
(You may answer yes or no) : NO
("Let's say no. But maybe I'll change my mind later...")

At what points during the process of calculating the Bode plot do you wish to have the opportunity to interrupt me :

1. at all possible points
2. I don't care, just give me the Bode plot
3. I want to have the opportunity to interrupt you at the point where you examine individual factors for their interesting frequencies
4. I want to have the opportunity to interrupt you when there is more than one factor causing the same break point
5. I want to have the opportunity to interrupt you when you are about to examine a new break frequency

Make a choice : 2

Is there anything you want to ask?
(You may answer yes or no) : WHY LEVELS 3
("What do you mean, and where are we in the program?")

Your answer supplies a value for the assertion <any_questions>
the user wants to ask questions

The assertion <any_questions> is a precondition of the action <interrupt_process>

I am about to proceed on the instructions you gave me. As long as you do not answer this question, you can interrupt my line of thought, e.g. for asking ?<name> questions.
in consultation area <get_FTF>
how to obtain the frequency-transfer function (FTF)

Please reply : ??
("What was the question again?")

Is there anything you want to ask?
(You may answer yes or no) : ?

I mean, is there anything you want to ask before I go on trying to get the FTF
Please reply : ?
("can you give me <u>even more</u> information on this question?")

Is there anything you want to ask?
(You may answer yes or no) : NO
(By asking the same question again, the program shows it has no more information)

(In the mean time, the user has changed his mind about not wanting extra explanation on the factors of the FTF. He'll use the next question as an interrupt point to change his answer.)

Is the frequency-transfer function available?

(You may answer yes or no) : LIST GOALS
("Let me see what the program has done, and what it is doing right now")

****

The following goals are under consideration :

Assertion <FTF_given> Not evaluated
Prior Value : -100.0
the frequency-transfer function is given

which is being considered because it is the precondition for action <read_given_FTF> ---------------

Area <get_FTF> : Status: now being investigated
how to obtain the frequency-transfer function (FTF)
contains the actions:

1/
I am about to proceed on the instructions you gave me. As long as you do not answer this question, you can interrupt my line of thought, e.g. for asking ?<name> questions. <interrupt_process>
2/ if the FTF is given, I will now read it <read_given_FTF>
3/
I am sorry, but I could not correctly interpret the FTF you gave me. Please try again and make sure it is in the right form. If you do not know what the right form is, examine FACTORS how_to_read_FTF <failure_to_read_FTF>
4/ if I had the knowledge, I would get the FTF in other ways when it is not directly available <more_actions_get_FTF>

which is being considered because it appears in action <Bode_plot>

Area <control> : Status : to be investigated
the overall sequencing
contains the actions:

1/ I will now calculate a Bode plot <Bode_plot>

which is being considered because it is the model's FIRST AREA.

****

The following goals have been completed :

Area <user_wishes> : Status : investigated
what advice the user expects from me
contains the actions:

1/ I give the inexperienced user some advice <familiar_with_SAGE>

2/ I set some of the features which will be used during the calculation of a Bode plot, if wanted <help_Bode_plot>

Assertion <xtra_xplanation_want> Current Value : -100.00
Prior Value : -100.00
you want extra explanation

Assertion <interrupts_possible> Current Value : -100.00
Prior Value : -100.00
the user has the opportunity to interrupt the thought process at appropriate moments

****

Please reply : !
("I will interrupt the normal course of the program, because I want to change the value of <xtra_xplanation_want>, which is now FALSE")

Please type a command : SET XTRA_XPLANATION_WANT

How certain are you that you want extra explanation?
(Range is -5 -> 5) : 5

<xtra_xplanation_want> has been SET to 100.00
(Notice how the certainty value "5" has been transformed into "100.00", which is "TRUE")

Please type a command : RESUME
("You may now continue with the normal course of the program")

Is the frequency-transfer function available?
(You may answer yes or no) : INVESTIGATE
("I wonder what happens if I give some answer rather than another. Let me INVESTIGATE that")
That command is not currently allowed
- type HELP for legal functions.
("Oh well, it seems I have to change to command mode again")

Please reply : !

Please type a command : INVESTIGATE

Is the frequency-transfer function available?
(You may answer yes or no) : NO
("Let's see what happens if I say NO". Notice how the same question is asked)

Since I do not yet have the knowledge to obtain the FTF in other ways, e.g. when the differential equation of the system is known, there is not much I can do.
For all frequencies, log|H| vs log(w) represents a straight line with slope UNKNOWN. |H| has the value 1.00 for w = 1.

*** That concludes the consultation. ***

Type QUIT to quit.

Please type a command : REINSTATE
(Notice how the program's answer didn't tell very much, in fact was completely wrong. Let's REINSTATE the old situation.)

Please type a command : RESUME
(RESUME the normal course of the program)

Is the frequency-transfer function available?
(You may answer yes or no) : YES

You may now type in the frequency-transfer function (FTF):
1/(jw)(1 + jw)?

For low frequencies, the asymptote of |H| in a log|H| vs log(w) plot represents a straight line with slope -1.00. For decreasing frequency, log|H| will approach the asymptote from below.
Above w = 1.00 the behavior of the asymptotes of |H| and arg(H) changes due to factor FACTOR-2. This factor, which is in the denominator of the FTF, represents a normal first order system. Below the break frequency, the gain of this factor has a horizontal asymptote and the phase tends to 0.00 degrees, above it the gain has an asymptote in a direction with tangent -1.00 in a log|H| vs log(w) plot, and the phase tends to -90.00 degrees. The previous asymptote intersected the break point frequency at |H| = 1.00, but the actual value of |H| at the break point is :
0.71.

The effect from this break point on the total asymptote of |H| is, that the slope changes to -2.00.

*** That concludes the consultation. ***

(Well? Did you see the extra explanation?)

S3. Dialogue 3

(In which the user from dialogue 2 continues, but now probes for the deeper levels of the program and more task-specific information).

Please type a command : BEGIN
(Now the program will start running again)

In this version of <Dynamic_Systems> the only thing I can do is calculate a Bode plot.
-- more (Y/N) ? NO

Do you want me to calculate a Bode plot?
(You may answer yes or no) : TRACE 1
(Now the program will show when it enters or leaves an area in the model)

Tracing level is now 1

Please reply : YES
(This is the answer on the last question)

We are considering what advice the user expects from me
(Intelligent readers of this report remember from the previous dialogue that this is the first action effect of ACTION <Bode_plot>!)

Are you familiar...*(We will skip printing the next three questions in this dialogue, since they are not interesting any more. See dialogue 1 if you want. Here the user will give the same answers as in that dialogue. We pick up the conversation on:)*

Make a choice : 1
("I'd like to be able to interrupt the program at all possible points")

We have completed considering what advice the user expects from me

We are considering how to obtain the frequency-transfer function (FTF)

Is there anything you want to ask?
(You may answer yes or no) : NO

Is the frequency-transfer function available?
(You may answer yes or no) : YES

You may now type in the frequency-transfer function (FTF):
(1 + jw tau)/[(jw tau -4)**2 + 3**2]?
(Notice that we use a variable "tau" in this FTF!)

We have completed considering how to obtain the frequency-transfer function (FTF)

We are considering initialization of increments and background memory space necessary for area <xmine_low_freq\break>

We are considering investigation of the low frequency behavior of the gain |H| and phase arg(H) for the FTF H, and preparation of a list of frequencies where a change in the behavior of |H| or arg(H) will occur

I have just completed investigating the first factor.
Is there anything you want to ask?
(You may answer yes or no) : TRACE 3
("Well apparently this is the first interrupt point in the area where the program investigates the low frequency behavior. Please give me full details on how you will go on: tracing level 3")

Tracing level is now 3

Please reply : NO
("I have no more questions, thank you. Please continue")

Node <ask_any_questions_4> evaluated to -100.00
Node <any_questions_4> evaluated to -100.00
****

Result for <any_questions_4>:
The likelihood that the user wants to ask questions is -100.00

****

The goal we are considering is : there is another factor of the total FTF to be investigated

Node <another_factor> is not yet evaluated
Node <check_next_factor> is not yet evaluated
Node <not_finished> evaluated to 100.00
Node <check_next_factor> evaluated too 100.00

Node <another_factor> evaluated to 100.00
****

Result for <another_factor>:
The likelihood that there is another factor of the total FTF to be investigated is 100.00

****
(Aha! Notice how the result of <u>goals</u> is put between those stars)

The goal being investigated is unnamed (it is a parameter or a precondition of an action)
(This is not very clear. But now the user unfortunately has no way to

interrupt the program to ask what is meant by this!)

Node <trivial_factor> is not yet evaluated
Node <check_trivial_factor> is not yet evaluated
Node <type> is not yet evaluated
Node <factor> is not yet evaluated
Node <obtain_factor> is not yet evaluated
Node <next_factor> is not yet evaluated
Node <FTF> evaluated to position POLYNOMIAL-1 (71.00)
Node <next_factor> is not yet evaluated
Node <factor_cycle> evaluated to position CYCLE-3 (3.00)
Node <next_factor> evaluated to 17.00
Node <obtain_factor> evaluated to 17.00
Node <factor> evaluated to FACTOR-2 (17.00)
Node <type> evaluated to 3.00
Node <check_trivial_factor> is Not Evaluable
Node <trivial_factor> evaluated to -100.00
****

Result for the unnamed goal is -100.00

****

The goal being considered...(*This monologue by the program will go on for several pages, until the next interrupt point, which is when the program has finished investigating the second factor of the FTF for its low frequency behavior. We will not print all, but pick up on:)*

****

Result for <value_factor_cycle>:
the number of factor cycles examined sofar is second (2.00)

****

having examined a factor I allow the user to interrupt my line of thought, if wanted

I have just completed investigating the second (2.00) factor.having examined a factor I allow the user to interrupt my line of thought, if wanted (*This is again a point where SAGE is not very clear, I think. But do you notice that this text "I have just.." is almost identical to the output text just* <u>*before*</u> *we changed the TRACE level?)*

The goal we are considering is : the user wants to ask questions

Node <any_questions_4> is not yet evaluated

Is there anything you want to ask?
(You may answer yes or no) : TRACE 0

("How strange! I have seen the same question and node <any_questions_4> before! The area <xmine_low_freq\break> probably uses RESTART to examine each new factor. But now I shall put the tracing level back to normal.")

Tracing level is now 0

Please reply : NO

For low frequencies, the asymptote of |H| in a log|H| vs log(w) plot represents a horizontal line. For decreasing frequency, log|H| will approach the asymptote from below.
Above w = 1.00* 1/tau the behavior of the asymptotes of |H| and arg(H) changes due to factor FACTOR-1. The previous asymptote intersected the break point frequency at |H| = 0.04, but the actual value of |H| at the break point is :
1.00SQRT[1 +1.00 tau**2]/SQRT[(25.00 -1.00 tau**2)**2 + 64.00tau**2].

The effect from this break point on the total asymptote of |H| is, that the slope changes to 1.00.
I have just completed investigating the first break point.
Is there anything you want to ask?
(You may answer yes or no) : NO
("Again an interrupt point!")

I will now examine the next interesting frequency.

Above w = 5.00* 1/tau the behavior of the asymptotes of |H| and arg(H) changes due to factor FACTOR-2. The previous asymptote intersected the break point frequency at |H| = 0.20, but the actual value of |H| at the break point is :
1.00SQRT[1 + 25.00 tau**2]/SQRT[(25.00 - 25.00 tau**2)**2 + 1600.00 tau**2].

The effect from this break point on the total asymptote of |H| is, that the slope changes to -1.00.
I have just completed investigating the second break point.
Is there anything you want to ask?
(You may answer yes or no) : NO
(Notice that here also the text is almost the same.)

*** That concludes the consultation. ***

Type QUIT to quit.

(Instead of quitting, the user wants to know, why log|H| will approach the asymptote from <u>above</u>. By doing LIST DONE GOALS and LIST DONE OBJECTS, which are similar to the LIST GOAL from the previous dialogue, he finds out that OBJECT <relative_position> is what he wants to see. We

left out that LIST commands in this dialogue, because their output needs a lot of space)

Please type a command : LIST RELATIVE_POSITION

Object <relative_position>   Current value : above
Range   (-10000.00E30 to 10000.00E30)   Scale : <position_scale>
relative position of |H| with respect to the asymptote of |H|
(Notice that no numerical value is visible, only the scale value "above"!)

Please type a command : FACTORS RELATIVE_POSITION
("How did you get the value "above" for <relative_position>?")

The object:
relative position of |H| with respect to the asymptote of |H|
<relative_position>
may be determined by:
**   1. using the rule <calc_rel_position>
(The 2 stars in front indicate that the rule has been used)

Please type a command : FACTORS CALC_REL_POSITION
("What does this rule do?")

By rule <calc_rel_position> the conclusion:
relative position of |H| with respect to the asymptote of |H|
<relative_position>
is:
coeff_1st_term_|H| * coeff_2nd_term_|H|
(because this product of coefficients is the actual coefficient of w**2 in the expansion of |H| for low frequencies, and therefore it determines the upward or downward parabolic direction)

Please type a command : FACTORS COEFF_1ST_TERM_|H|

The object:
the coefficient of the first term of the expansion of the total |H|
<coeff_1st_term_|H|>
may be determined by:
1. using the rule <calc_coeff_1st_term1>
2. using the rule <calc_coeff_1st_term2>
**   3. using the rule <calc_coeff_1st_term3>
4. taking the default <1.00>

(Now the user can go on asking what this rule <calc_coeff_1st_term3> does or why the others failed, but we'll leave that to you. Let's examine another object, whose name you have already seen several times:)

Please type a command : FACTORS BREAK_FREQUENCY LEVELS 2

The object:

the break frequency <break_frequency>
may be determined by:
** 1. using the rule <get_breakfrequency>

By rule <get_breakfrequency> the conclusion:
the break frequency <break_frequency>
is:
current_element(list, break_point_cycle)
(we take an interesting frequency from our list of interesting frequencies)

Please type a command : ?list
(*instead of asking how you should determine <list> or the other argument of the FUNCTION <current_element>, we ask what it is*).

the position of a list of all break points
Please type a command : ?
(*"Have you got more information on that?"*)

a list of all the points where a change in the behavior of gain |H| and phase arg(H) will occur.
Please type a command : ?

The actual value indicates the first position in background memory where the list is kept.
Please type a command : ?

Sorry - no further explanation available.
(*Readers who are wondering how this break frequency got into that list, are advised to do HELP BREAK FREQUENCY.*
But seriously, this is done in area <xmine_low_freq\break>).

Please type a command : QUIT
(*This is the end of this dialogue*)

QUIT - are you sure ? (Y or N) YES

*** SAGE consultation finished. ***

>

S4. Dialogue 4

(Suppose the user from dialogue 1 decides not to quit, but to find out what would happen if he continues, and introduces a variable in the FTF).

Please type a command : BEGIN
(Now the program will start running again, but background memory is unchanged)

In this version of <Dynamic_Systems> the only thing I can do is calculate a Bode plot.
-- more (Y/N) ? NO

Do you want me to calculate a Bode plot?
(You may answer yes or no) : !
(Since part of the work has already been done in dialogue 1, like getting the FTF, the user will now take control of the course of the program, by changing to command mode "!")

Please type a command : FORGET ALL
(The program should forget the normal course of events)

All goals forgotten.

Please type a command : CONSIDER EXAMINE_INTERVALS

The area <examine_intervals> will be considered immediately.
(However, the program will do nothing until its course is RESUMEd)

Please type a command : CONSIDER XMINE_LOW_FREQ\BREAK

The area <xmine_low_freq\break> will be considered immediately.
(And after that, <examine_intervals>!)

Please type a command : CONSIDER PREP_LOW_FREQ\BREAK

The area <prep_low_freq\break> will be considered immediately.

Please type a command : SET VARIABLE

Is the variable of a factor

1.
2. tau
3. tau_1
4. tau_2
5. tau_3
6. t
7. t_1
8. t_2

9. t_3
10. unknown

Make a choice : 3

<variable> has been set to tau_1

Please type a command : RESUME
(Now the program should execute the areas in the order specified)

At what points during the process of calculating the Bode plot do you wish to have the opportunity to interrupt me :

1. at all possible points
2. I don't care, just give me the Bode plot
3. I want to have the opportunity to interrupt you at the point where you examine individual factors for their interesting frequencies
4. I want to have the opportunity to interrupt you when there is more than one factor causing the same break point
5. I want to have the opportunity to interrupt you when you are about to examine a new break frequency

Make a choice : 2
(Notice that this question is part of the area <user_wishes>, which has been set to unevaluated by the BEGIN command. This area is not among the ones the user ordered to CONSIDER, so assertions (and questions) from this area will be examined only if, and when they are needed by the other areas)

For low frequencies, the asymptote of |H| in a log|H| vs log(w) plot represents a horizontal line. For decreasing frequency, log|H| will approach the asymptote from below.
Above w = 0.33* 1/tau_1 the behavior of the asymptotes of |H| and arg(H) changes due to factor FACTOR-1.
When I give you the Bode plot, do you wish me to explain in more detail the contribution of each factor?
(you may answer yes or no) : NO
(Notice again, that now this question is asked when it is needed!)

The previous asymptote intersected the break point frequency at |H| = 1.00, but the actual value of |H| at the break point is :
0.71.
*(This is wrong. It should be 1.00/SQRT[1 + 1.00 tau_1 **2]. This error is due to the fact that in the background memory the variable has not been set, but only in the SAGE model part)*

The effect from this break point on the total asymptote of |H| is, that the slope changes to -1.00.

*** That concludes the consultation. ***

Type QUIT to quit.

Please type a command : SET VARIABLE
("What if I set <variable> to another value?")

Is the variable of a factor

1.
2. tau
3. tau_1
4. tau_2
5. tau_3
6. t
7. t_1
8. t_2
9. t_3
10. unknown

Make a choice : 4

<variable> has been SET to tau_2
%RECORD ACCESS ERROR 2 (type mismatch) 14 2346 1 0

Fatal error number 13 GETACS/AxsErr
>
("Hm, I seem to have hit a sore spot. Well, that's the end of our conversation.")

Chapter 4
Analysis of KNOWBODE

This chapter may supplement the knowledge on "what KNOWBODE does" gained in the last chapter, with knowledge "how KNOWBODE does it". This is a very tedious chapter.
It just provides you with some figures on the details of KNOWBODE, which is necessary because in subsequent chapters design problems are illustrated using examples from KNOWBODE, and you may need this background information.
I suggest you glance through this chapter, and return to it when needed.

A slight attempt is made to compare KNOWBODE with R1 (currently called XCON) by McDermott ([7]), an attempt doomed from the start because KNOWBODE is too small.

§1. The structure of dialogues in the SAGE system

The greatest part of the dialogues with the SAGE system is a program-initiated question answering process. This is often considered too rigid a structure (see [6]), and that was in fact one of the reasons why I decided to make a FUNCTION to take care of direct input (<read>, see Appendix C). But the fact that SAGE is a backwards chaining system makes sure that focused questions are asked: see for example dialogue 4, where SAGE asks the questions when they are needed (see also Winston, [19] p.199). Since SAGE is based on goal trees (and AREAs and ACTIONs also form a strict hierarchy), "how" and "why" questions are possible ([19], p.41).
When giving answers and explanation to the user, SAGE uses canned sentence fragments, and the text associated (by the designer) with the elements of the model. SCALEs may be used to transform numerical output into words, and allow the user to make a choice between words on input (see dialogue 4 on SET <variable>). However, there again it is the system which must name the objects. Since the user must find out what a certain object is called by the system, the program should sometimes identify it in the dialogue: " ....due to factor FACTOR-2." This was also observed on MYCIN ([19],p.197).
More advanced types of discourse in SAGE are theoretically possible, but they would require expert knowledge on the rule systems that govern person-person communication (see [13]) and would probably obscure the (actual) expert task.

One may add, that due to truth-maintenance (the value of ASSERTIONS/OBJECTS remains unchanged when evaluated) coherence throughout the dialogue is maintained. In a normal run, SAGE will not ask the same question twice.

See also Chapter 8, §4.4 (QUESTIONS).

§2. The model

As described in Chapter 1, a model consists of an ACTION-based part and a RULE-based part. We shall describe both of them in the case of KNOWBODE.

§2.1. The ACTION-based part of KNOWBODE

The ACTION-based part consists of a hierarchy of AREAS (see fig. 7). These AREAs contain ACTIONs that may call other AREAs, initiate investigation of goal trees and take care of output. In fig. 8 we indicated for all AREAs the number of ACTIONs, action effects, the number of different ASSERTIONs/OBJECTs which appear in the ACTIONs of that AREA (even as part of a function/procedure call) and we indicated the number of RULEs/ASSERTIONs/OBJECTs/QUESTIONs declared in that AREA.

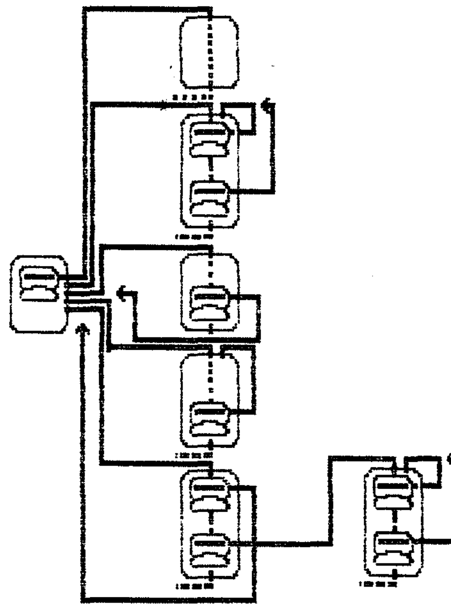


fig. 7 a hierarchy of subtasks

	ACTIONs	action effects	ASSERTIONS or OBJECTs called	RULEs	ASSERTIONs	OBJECTs	QUESTIONs
control	1	2	1	2	2	2	1
user_wishes	2	4	4	-	6	-	3
get_FTF	4	5	3	1	3	-	2
prep_low_freq\break	2	2	1	1	1	1	-
xmine_low_freq\break	11	16	22	28	9	23	1
examine_intervals	2	3	2	-	-	1	-
body_xmine_intervals	9	16	23	29	4	25	2
	31	48	(49)	61	25	52	9

fig. 8 first 3 columns indicate ACTIONs, action effects and (different) facts investigated in AREAs, last 4 columns indicate RULEs, QUESTIONs, ASSERTIONs and OBJECTs declared in the AREA.

The total number of ACTIONs + RULEs (92) may be taken as a measure of the size of the task. Compared with other expert systems, like R1 ([7]), which has 772 rules, KNOWBODE is small (which was to be expected). There is not much more to say.

Of the 48 action effects, 16 take care of output (ADVISE) and 11 of the explicit flow of control in and between AREAs (RESTART, REASK, CONSIDER <area> and FORGET <area>).

49 different ASSERTIONs and OBJECTs appear in the ACTIONs. As already stated in the footnote on p. 10., I would like to call these facts goals. Viewed from the AREAs, the RULE-based part consists of a forest of 49 goal trees.
Comparing this number with the total number of facts (77), creates the suspicion that there is a shallow forest (i.e. a forest where the root nodes are only a few branches away from the external environment) or a very bushy forest (there is a tangled structure among the trees) but in any case, the AREAs are not far from the external environment.

The fact that the number of OBJECTs is twice as large as the number of ASSERTIONs indicates that this is a numerical task.

If you investigate the nature of the questions asked in the dialogues, you see that they are only concerned with what kind of output the user wants. Apart from the FTF, no additional (domain-specific) information has to be specified by the user.

§2.2. The RULE-based part of KNOWBODE

If we examine the RULEs of KNOWBODE in terms of the functions that they perform and the extent to which they embody domain-specific knowledge, we find that only approximately 27 RULEs (of the 61) may be considered directly related to knowledge about Bode plots. An example of such a RULE may be found on p. 9: RULE <calc_break_point_2>.
24 RULEs are concerned with retrieving data from the external environment, or storing data there. An example is <get_break_frequency> on p. 31.
The rest (10 RULEs) may be considered "neutral" and have a function somewhere in between*. Only one of them is considered a pure computational rule:

```
RULE convert_to_|H| :
"conversion of log(|H|) back to |H|"
        intersect_asym_|H| IS
                expon( 10., log_intersect_|H|)
```

Comparing the number of storage/retrieval RULEs with domain-specific ones indicates that KNOWBODE relies heavily on data from the external

* I shall not print them here, but e.g. RULE <check_horizontal> and RULE <check_1_break_point> are both considered neutral RULES by me.

environment, which is also confirmed by the fact that only 11 RULES contain no FUNCTION whatsoever.
By the way, in R1 the number of domain-specific rules is 480 opposed to 292 more general ones.

That part of a goal tree which is not domain-specific I shall call the "grey zone" (or "twilight zone"?). It may be called grey, because it is not interesting to the end-user (and should therefore be invisible), but is to the designer (who must be able to inspect and understand it). It is somewhere in between the external environment (obscured, black) and the high-level part of the model (visible, white). In Chapter 9 I shall say some more on this.

If we examine the <u>number of RULES or QUESTIONS available to establish a single fact</u>, we find:
2 OBJECTS (<|H|> and <arg(H)>) which cannot be evaluated because there are no DEFAULT values, RULES or QUESTIONS to establish them. They are used only as a "hidden HELP option", for the user may ask their associated text (explanation) by using the "?" command.
15 OBJECTS which have only a DEFAULT value and no RULES or QUESTIONS to establish them. They are used to indicate memory positions in background memory, necessary as arguments in FUNCTIONS and PROCEDURES. Since they are OBJECTS rather than CONSTANTS, they may have (explanatory) text associated with them. An example: <list>.
50 ASSERTIONS or OBJECTS have only 1 RULE or QUESTION to establish a value for them, and possibly a DEFAULT value. Only one of these 50 is ASKABLE.
7 ASSERTIONS or OBJECTS have 2 RULES or QUESTIONS to establish a value for them (and possibly a DEFAULT value).
2 ASSERTIONS or OBJECTS have 3 RULES or QUESTIONS to establish a value for them (and possibly a DEFAULT value).
1 OBJECT (<slope_|H|>) has 4 RULES to establish a value for it.
There are no ASSERTIONS or OBJECTS which have more than 4 RULES or QUESTIONS to establish a value for them.

The measures of performance for searching a goal tree, as mentioned in [9], p.91-94, cannot be properly used for evaluating SAGE.
However, I shall try to give a few criteria which indicate something of the nature of the KNOWBODE task.
1) The <u>number of branches</u> at each fact node in the tree may be considered a measure of the difficulty of the task*. This number is determined by 2 aspects:

a) the number of RULES or QUESTIONS available to establish a single fact. As you have seen above, this is about 1 RULE or QUESTION per fact: this means that usually the situation is specific** enough to determine the value of that fact.
b) the number of different ASSERTIONS or OBJECTS in the

*** See [7]: The VAX configuration task, which is performed by the expert system R1 is nontrivial because the number of <u>system variations is so large</u>.**

preconditions or body of the RULES (or in the preconditions of QUESTIONS). This is an indication how "specialized" the situation is in which the fact may be evaluated by that RULE.
Varying over all RULES, we find an average of 1.7 different ASSERTIONS or OBJECTS in the preconditions or body of RULES. If we include in the calculation the number of FUNCTIONS with different arguments appearing in the RULES, we find an average of 2.8 nodes per RULE. Actually, this number should be considered higher, because so many of the RULES only serve to retrieve a value from the external environment. In [7] (p.65) a somewhat similar measure for the expert system R1 is mentioned, ("fan-out"), with a value of 8.

2) the "bushiness" of the trees may be considered a measure of the difference in knowledge needed to establish a fact. It is determined by the number of RULES that use the same fact (or the same call of situation of a FUNCTION), averaged over all facts. I did not compute that, but do mention that the node "type(factor)" is used in 17 different places in the KNOWBODE model. SAGE is well equipped to handle multiple calls on tthe same node, because it will compute such a node only once!

3) the depth of the goal tree may, in combination with the number of branches on each node, be considered a measure for the amount of effort a program has to spend before it will attain a goal. But if there is a very bushy forest, one shouldn't forget that SAGE needs not examine a (partial) tree twice.

An upper bound of the minimum "distance" of a fact to the external environment in KNOWBODE is 6 RULES, (for <slope_|H|>). This means that, if they were ordered properly, SAGE would have to examine a sequence of 6 RULES before it retrieved any value for a precondition from the external environment. Usually, the forest is much shallower: every node needs to be expanded only a few levels before any of its preconditions is evaluated.

In KNOWBODE we have a shallow, bushy but simple forest, corresponding to a task which depends heavily on the External Environment, is based on a small amount of related knowledge and considers only simple situations.

§3. The interface and external environment

§3.1. The interface: FUNCTIONs and PROCEDUREs

The FUNCTIONs and PROCEDUREs which make up KNOWBODE's interface to the external environment may be divided into 4 classes:

storage/retrieval

first_element	2
current_element	2
next_factor	1
not_finished	3
cause	1

examine	1
put_in_memory	2
add	1
increase	3

recorded	10
put_on_record	5

type	2
get_alpha	1
get_beta	1
get_x	1
get_num_or_denom	2
power	1
mu	1
get_variable	2

direct in/output

value_|H|	1
read	1

computation

square_root	1
expon	2
log	3

internal

remove	1
rename_factors	1
delete	1
clean	1
sort	1

fig. 9 FUNCTIONs/PROCEDUREs in KNOWBODE. Figure indicates number of different call situations (a situation with different call parameters) in KNOWBODE.

N.B. The number behind the FUNCTION or PROCEDURE does not indicate how many times the FUNCTION or PROCEDURE was called, since due to RESTART any function or procedure may possibly be called as many times as there are factors in the FTF, to get new data from the external environment. It is more an indication of how general they are.

Functions which take care of *storage* or *retrieval* are abundant in KNOWBODE. The most basic storage/retrieval functions are <recorded> and <put_on_record>. As you can see, they are called in several situations (i.e. calls with different parameters) which means they are the most general functions.

The FUNCTIONs <type>...<get_variable> all retrieve a different aspect of a factor of the FTF. The fact that <type>, <get_num_or_denom> and <get_variable> have 2 call situations is a result of the fact that ASSERTIONS/OBJECTS declared in an AREA different from the AREA in which they are used, are not set to unevaluated when the AREA in which they are used is RESTARTed. Thus, both in AREA <xmine_low_freq\break> and <body_xmine_intervals> "type(factor)" should be called, but we need to introduce an extra object <factorX> in one AREA to make that possible.

The FUNCTIONs <first_element>...<cause> retrieve elements from a list (i.e. a polynomial or a list of break frequencies). They keep track of the elements either by an increment (one of the calling parameters indicating a memory position where this increment is stored) or by comparing the argument with elements in the first column of a two-dimensional list and returning an associated parameter from the second column of that list (<cause>).

<examine>...<increase> are FUNCTIONs and PROCEDURES which take care of storing or adding elements to a list or a specific memory position. Memory positions may either be explicitly mentioned in the arguments (used in <add>, <put_in_memory> and <increase>) or implicitly: used in <examine>. (The use of different memory positions explains why there are different call situations).

<current_element>/<next_factor> and <get_num_or_denom>/<power> are FUNCTION pairs which act on the same external routine. Use of different names was made for better understandability : see Chapter 9.

Direct input and output, i.e. directly from the terminal to the external environment is performed by <read> and <value_|H|>.

Computational procedures are used very little in KNOWBODE: <square_root>, <log> and <expon> are the only ones.

Internal procedures are PROCEDURES which have only effects inside the external environment (mostly clearing memory positions).

§3.2. The External Environment.

The actual routines which realize the FUNCTIONs and PROCEDUREs, are FORTRAN programs. A functional dependency scheme of those routines is included in part 3. <value_|H|> and <read> call upon a tree of FORTRAN routines, but most other functions and procedures are small and simple pieces of program included in a FORTRAN routine called XXSWIF (which is the basic external program called by SAGE).

The memory function of the external environment, or background memory as I call it, is realized by a single REAL ARRAY which currently has 200 positions. The virtual organization may be considered as in the following figure:

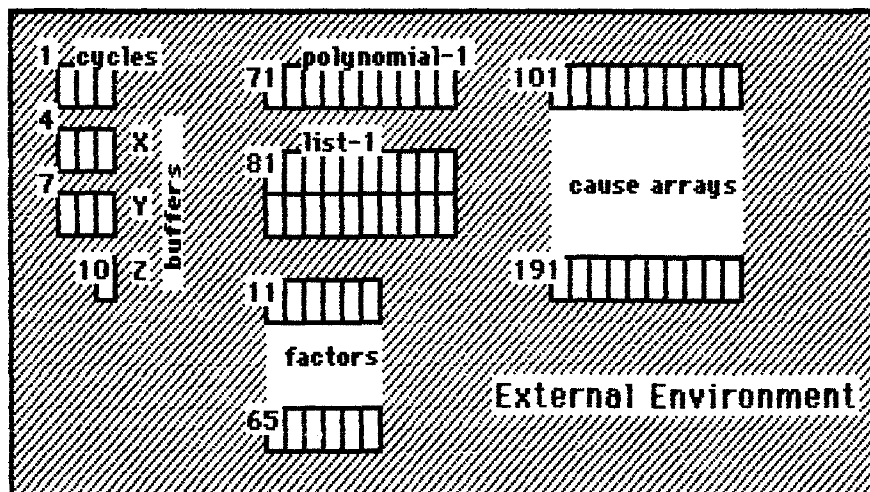


fig. 10 (Virtual) organization of background memory.

Background memory contains 4 types of data:

1) increments (in CYCLE-1 through 3), necessary to keep track of cycles executed, see Chapter 10.

2) intermediate results of calculations (BUFFER-X, Y and Z), see also Chapter 10.

3) data on the FTF (in POLYNOMIAL-1 and the FACTOR arrays). Elements of POLYNOMIAL-1 are first positions of factor arrays.

4) data on the break points of the FTF, and the factors that caused them (in LIST-1 and the CAUSE arrays). The first row of LIST-1 contains break frequency values, the second row first positions of cause arrays where the factors causing a break point are stored. In the cause arrays, first positions of factor arrays are stored (see [11] and Appendix D).

§4. Speed and size

In the next figure the measured time span between a user's answer and the subsequent question of KNOWBODE is displayed for dialogue 1. Figures between brackets correspond to points in the conversation. At point (18) KNOWBODE starts the output on the Bode plot. Points (18) to (22) correspond with the beginning of lines displayed on the terminal*. At those points in time a few new lines of output were presented.

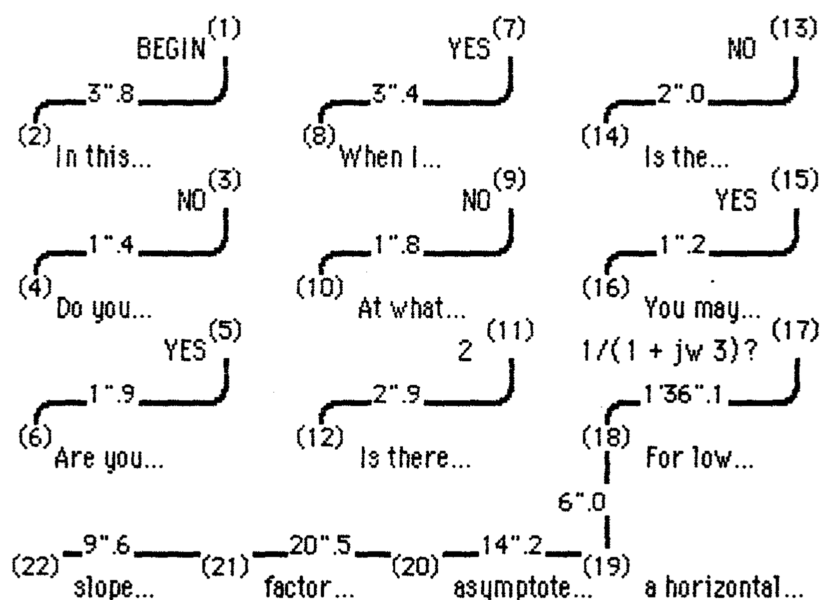


fig. 11 time span between user's answers and KNOWBODE's questions in dialogue 1

We may use the data in this figure to give a rough estimate of the speed of KNOWBODE**. In the next figure I have indicated the number of AREAs entered, ACTIONs entered (i.e. where preconditions succeeded) and RULES and QUESTIONS processed*** in each time span. Since the inner mechanisms of SAGE are unknown to me, I am not sure these are the

*** Due to technicalities it was not possible to type the lines in the dialogues in this report in the same length as in reality.**

**** KNOWBODE is run on a PDP-11/34 under RSX-11M, with the details (pages of core etc.) as indicated in file SAGEE.CMD which is included in part 3 of this report**

***** Since a lot of work in investigating a RULE is done in investigating the preconditions I treated RULES that failed or succeeded in the same way.**

factors that determine the speed of the program. Using the data between point (5) and point (16) from these two figures, we may roughly calculate that the time SAGE needs to enter an AREA is ± 1 sec., to enter an ACTION also ± 1 sec. and the time to process a RULE or QUESTION ± 2 seconds. Using this as a prediction for the time needed between (17) and (18) we would find ± 47 sec., whereas the actual duration is twice as large! As a possible explanation I point out that RESTART is used in this interval, which in TRACE 3 mode consultations was observed to take a long time.
As a general conclusion I may add that the system response is acceptable in a "conversation", when the number of RULES to be used is small, but slow when more RULEs or QUESTIONs have to be processed.

	(1) → (2)	(3) → (4)	(5) → (6)	(7) → (8)	(9) → (10)	(11) → (12)	(13) → (14)	(15) → (16)	(17) → (18)
AREAs entered	UNKNOWN	1	1	-	-	1	-	-	2
ACTIONs entered		-	1	2	-	-	-	1	9
QUESTIONS/RULES processed		1	1	1	1	1	1	-	18

fig. 12 Notice that 1) the AREA entered in (3) -> (4) is FIRSTAREA (and therefore left out of the calculation) 2) times are not interesting above point (18), because SAGE waits till an output buffer is filled. See part 3, "direct input/output".

The size of the SAGE Executive is 31,456 words, which is quite large. The compiled version of the model KNOWBODE is 415 blocks, about 104 K, which doesn't tell me much because I don't know how SAGE encodes the model (e.g. is the heavy use of RESTART, which KNOWBODE makes, a problem?). But compared to the total number of RULES and ACTIONS it seems quite large. The size of the external environment is also large, and FUNCTIONs and PROCEDUREs had to be put in an overlay structure (see part 3).

Chapter 5
The representation problem

A *representation* is a set of syntactic and semantic conventions that make it possible to describe things. The *syntax* of a representation specifies the symbols that may be used and the ways those symbols may be arranged. The *semantics* of a representation specifies how meaning is embodied in the symbols and the symbol arrangements allowed by the syntax. (Winston,[19] p.254).

The SAGE modelling language is part of a representation for expert tasks. The syntax of that representation is described in [21]SAG02, [22]SAG03 and Appendix A of this report*. As far as the semantics is concerned, the naïve user might get the idea that it is almost the same as the semantics of some subset of a natural language: ([20]SAG01 p.12:) "SAGE offers a simple but powerful language in which the knowledge of the expert can be encoded. It is not a programming language - for we are creating data and not programs - but more like stylised English."

However, this is not quite true. Knowledge, encoded in natural language (as in "The Expert Knowledge" on p.14) is usually enough for human beings to be able to solve questions in a specific problem area. How is that possible? What happens when we see or hear text in a familiar natural language? "We can think of all language use as a way of activating procedures within the hearer, any utterance as a program - one that indirectly causes a set of operations to be carried out within the hearer's cognitive system." (Terry Winograd, quoted by Hofstadter [4], p. 629).In terms of knowledge, we may say that <u>implicit</u> in the encoded knowledge is the knowledge that human beings have such and such capabilities, enough to make some sense of the words and sentences.

Compared to human beings SAGE has very limited capabilities. The SAGE modelling language syntax must help us to stylize our English into a form understandable for the SAGE system. But the syntax is "not restrictive enough" to permit only useful encoding of knowledge.

As an illustration of this, let us examine how a naïve user might encode the following sentence: "We note that if $wT \gg 1$, that is $w \gg 1/T$, then this term [i.e. of the form $\pm 10 \log[1 + (wT)^2]$] approaches the asymptote $\pm 20 \log(wT)$."(p.15).

He might encode it as

```
ACTION advice_term_type_4 :
        "Advice on terms of the form + or - 10 "
        "log[1 + (wT)**2]"
        ADVISE "We note that if wT >> 1, that is w >> 1/T, "
        "then terms of the form + or - 10 log[1 + (wT)**2] "
        "approach the asymptote + or - 20 log(wT)."
```

* Notice that syntax description may already include some semantics: e.g. using words as "AREA","ADVISE","actionname" tell you something of the purpose of the symbols.

Or as

```
ACTION advice_term_type_4 :
          "Advice on terms of the form + or - 10 "
          "log[1 + (wT)**2] for high frequencies"
          ADVISE "This term will approach the asymptote "
          "+ or - 20 log(wT)"
          PROVIDED type_of_term = plus_min_10_log_etc
                    AND wT_much_larger_than1
```

(plus appropriate encodings for assertions as <wT_much_larger_than_1>, and rules to establish those assertions)

or he might encode the knowledge in a RULE:

```
RULE check_bhav_trm_type4 :
          "checks behavior for type 4 terms for high frequencies"
          trm_appr_asym_20_log IS TRUE
          PROVIDED type_of_term = plus_min_10_log_etc
                    AND wT_much_larger_than1
```

(again plus appropriate encodings for the assertions named, and rules to establish those assertions, e.g.)

```
ASSERTION wT_much_larger_than1 :
" wT is much larger than 1"

RULE check_wT_much_larger :
"a simple test: depending on the constant <many_times> "
"the assertion is either true or false"
          wT_much_larger_than1 IS TRUE
          PROVIDED frequency_w > many_times *1 / T
```

It is obvious that there are many more ways to encode the same knowledge in the SAGE building blocks. Less obvious is the fact that most of those ways will yield useless programs, which at best give some advice on details of the problem, but do not solve the overall problem. Those programs lack a "sense of direction": they do not know what is "important", what the user wants.
We may say that the SAGE modelling language (as the natural languages in general) has not the right structure to <u>force</u> us to encode that sense of direction into the model. Also, the SAGE system is not powerful enough to generate that sense itself. I therefore propose to treat the SAGE modelling language as a <u>programming language</u>, to make us aware that the <u>knowledge engineers</u>, who put the knowledge into the program, must put more structure into that knowledge than they would when talking to some human being*.

*Consider Simon [16] p.27 : " in assessing problem solving procedures we must consider not only their performance in <u>solving</u> problems once these problems have been represented, but also their adequacy in <u>representing</u> problems when those problems are presented initially in natural-language prose". The term "problem solving" is used twice in this sentence: we can infer that people sometimes find it hard to distinguish between making+using a representation, and just using a representation. (Or, in terms of knowledge: between the knowledge needed to create a certain problem representation and the knowledge stored in that representation). See also part 1 of this report.

At the most abstract level the representation problem is in fact the design problem: how to make a good expert system that will do what we want it to do. Which brings us on the design criteria.

§1. Design Criteria

During the design of the expert system, I came to the conclusion that the design criteria may be more or less divided in three classes, which I will name*

[1] performance
[2] understandability
[3] flexibility.

I am aware that in several publications (other) criteria for program design have been formulated. The division which I describe here is an attempt to clarify the specific needs on designing a (SAGE) expert system.

§2. Performance

The most important criterion in designing a program is : when I take this piece of program, and put it into the program I'm trying to build, (how good) will it do the job? This is what I call the "performance" criterion.
"Performance" has to do with the details of how a procedure realizes a piece of knowledge.
For a procedure to be a good performance procedure, there must be a demonstration that competence (expert knowledge) has been put to use effectively ([19] Winston, p.295). This means, that I test whether the program does what I want it to do. And I test in what situations it does that.
During development of a program, the designer should test the performance by running the program, either in his mind or (actually) on the computer.

§3. Understandability

A second design criterion is the requirement that the expert system (as any computer program) must be as understandable as desired by two kinds of (intelligent) systems:

> 1. (anticipated) end-users. By "anticipated" we mean that the expert system can expect the user to have some knowledge of the subject on which he wants advice from the system. For instance, in KNOWBODE we expect the user to understand

* Although it proved possible to make the system without explicit formulation of the criteria, it may speed up design. But see also p for similar situations in designing dialogues. What is expressed here does not contradict my conclusion that design by prototyping is very useful when the necessary knowledge is not explicit.

english, and expect him to be able to give the frequency-transfer function. The designer of the system must decide what to expect from the end-user.
2. those who wish to (further) develop this program or a similar one, the designers.

Good understandability can be achieved by the following principle: "show the observer what he wants to see, hide from him what he doesn't want to see". Well, what does a user want to see? Results. But in order to reach those results, the system needs information from the user. However the user usually does not know what information the system wants and why it wants that, because an expert system is generally based on knowledge which the user lacks. But if it is possible for the system to show what it is trying to do, then maybe, if the user understands enough of it, it may provide some information, and together they can conduct some sort of "dialogue" to get the results.
In a "normal" computer program the user must try to understand what the program does by the output it gives. The designer may look at the listing. But what we actually want is, that what is visible of the program to be in "human" terms, on a level of abstraction which is common for people. Since the "human level" is not a fixed level of abstraction, there must also be a possibility to "translate" higher order concepts or descriptions into lower ones. As will be shown, the goal trees that SAGE provides allow such a chunked description, and allow introspective question answering ([19], p.40).

The procedure which SAGE uses to examine facts incorporates search. Different parts of the model will be used in several places of the program. It is therefore not very easy for the designer to inspect the model by looking at the listing! The designer must be regarded as a special kind of user, with different interests, and his primary means of testing the program is actually running it.

SAGE is not perfect. Sometimes the designer is forced to use certain constructions to realize a procedure which are not "proper", which are not as people would do. These constructions must be hidden from the user, but should be visible to the designer.

§4. Flexibility

In particular during design, (when you do not yet know what you're going to need) it is important that the way you represent your problem is flexible.
By "flexible" I mean easily changeable (when additional knowledge is incorporated).
For instance, the "flexibility" criterion is a reason why the knowledge encoded in AREA <user_wishes> (examination of the results the user wants) is not directly included in AREA <control> : in this way, an extended version of KNOWBODE which will be able to compute polar plots as well as

Bode plots, may use this AREA as an independent module*.

"Flexibility" may conflict with understandability: making ever more general structures to perform some task creates a haze about what is actually happening. In my treatment of the design of the External Environment, in Chapter 9, I will clarify this. In the end the performance of a program is determined by the flexibility of the representation (see Hofstadter, [4], p.296-302).

* Another argument is e.g. the "understandability" criterion : the purpose and functioning of <user_wishes> will be clearer if it is in a separate area instead of tangled up in <control>.

Chapter 6
How to solve a problem using the SAGE building blocks.

Here we will deal with the question on how to get good "performance" of the expert system. In our case, how can we let the system describe a Bode plot?
A powerful technique which was developed for converting global goals, like "making a Bode plot", into local strategies is called problem reduction (Hofstadter [4] p.609/610, Simon [15] p.73, Winston [19] p.33) or goal reduction. It is based on the idea that whenever one has a long-range goal, there are usually subgoals whose attainment will aid in the attainment of the main goal. Therefore if one breaks up a given problem into a series of new subproblems, and so on, in a recursive fashion, one eventually comes down to very modest goals which can presumably be attained in a couple of steps.
At this point I want to make two remarks. First, it may be clear that what human beings call "a very modest goal" may still be very difficult for a computer. And what some people call a very modest goal may be very difficult for others. There is not really a clear division. We will see that the backwards chaining that SAGE does will allow the user to find where his own "modest goals" lie. Maybe you saw that already in dialogue 3: if a user knows what "a coefficient of the first term of the expansion of the total |H|" is, he need not ask further questions. But if he doesn't know, he may ask them!
Second, there is no reason to expect that the decomposition of the complete problem into subproblems will be unique. The usefulness and understandability of "good" representations (i.e. structures (=programs) and processes (=SAGE) in which the problem can be solved) may vary enormously. When searching for a good representation, we should keep the specific strengths and weaknesses of SAGE in mind. What those strengths and weaknesses are, we will find out soon enough.

Let us* try to use problem reduction to make a structure in the SAGE modelling language, in which the problem may be solved. From what source do we get ideas on which subgoals may aid in attaining the main goal?
This is where the expert comes in. At some time in the past he/she has learned what steps to take in order to achieve the goal. The expert may not be able to tell you how to do it, but that's another thing entirely. By confronting a working system and an expert with real problem situations, we may remedy differences in performance, just as we may change the vocabulary and explanation used in confrontations with a real end-user (this is called *design by prototyping*).
In the KNOWBODE case the designer had to become an "expert" himself, by reading books like [3] and [12], and solving example problems.

*** People have proved to be able to solve problems (sometimes). At present, there are a few AI programs like "ISAAC" and "UNDERSTAND" which can translate (a few) problems posed in natural language into formal state-space representations (brief description included in [16] p. 23-25). Notice the difference with SAGE: at best you may say that SAGE is not an expert, but may become one (KNOWBODE), whereas those programs are experts in representing things, but cannot become experts in making a Bode plot.**

S1. How to make a Bode plot

As the last sentence of "the expert knowledge" (p.15) suggested, it seems a good idea if we could split the FTF into factors of some basic types, then examine the gain and phase for those types, and obtain the total gain and phase by simply summing up the parts. This would mean some decomposition like in this figure:

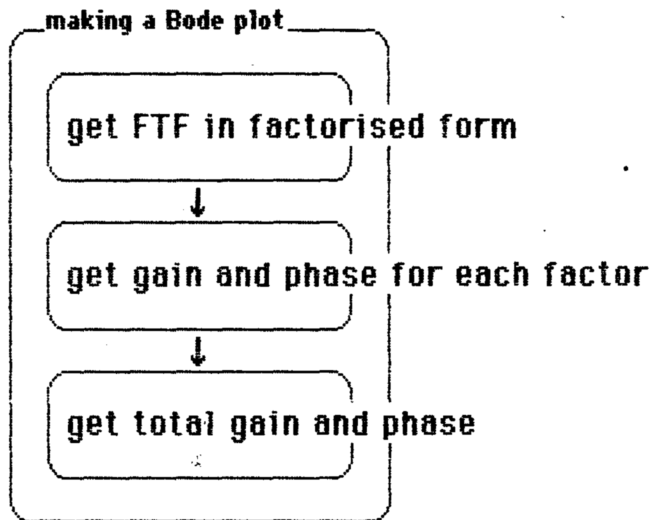
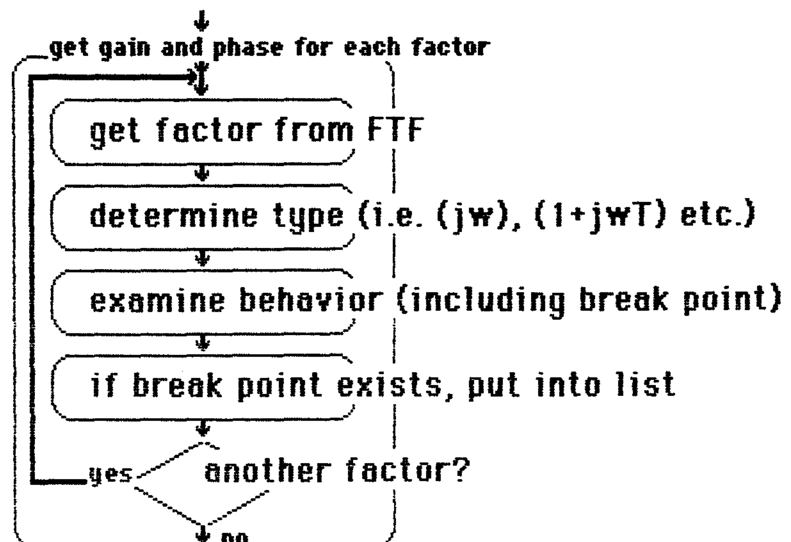


fig. 13 first decomposition of the problem

But the actual problem is not a matter of simple calculation: we are interested in the behavior of gain and phase. We want to see things like "[...] this plot approaches the asymptote ± 20 log(wT); which is a straight line with slope [...]"(etc.). The question is, whether this information may be "summed up" more or less in the same way the values for gain and phase are summed up.

If we know the break points of individual factors, i.e. the frequencies where there is a change in the slope of their asymptotes, we may calculate the total asymptote of |H| (in a logarithmic plot) in an interval between break points, by simply adding the slopes of all (individual) asymptotes in that interval. We cannot examine the break points in increasing order, if we don't know that order first. Therefore we may now give a possible decompositon of the subtask "get gain (and phase) for each factor":



Notice that we have introduced a loop to examine all factors. The list was necessary, because we first had to put the break points in the right order to be able to examine them for the behavior of the total plot. A part of the decomposition of the subtask that examines the total behavior of the Bode plot may now be:

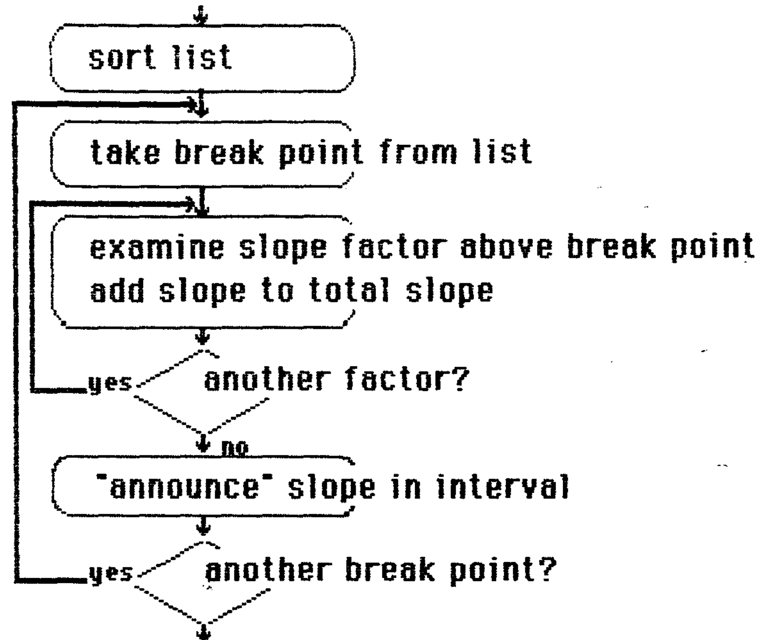


fig. 15 (part of) decomposition of the third subtask

Now we have two loops, one nested in the other. For every interval between break points, we have to examine the slope of all factors, and add them together. It took me quite some time to realize that this is not necessary! Since the asymptotes are all straight line segments, I only have to examine in each frequency interval factors for which the slope has changed, and only add the slope of the asymptotes of those factors to the total slope*.

The factors for which the slope has changed, are the factors that "caused" a new frequency interval (another break point) in the first place! So, if I remember which factor caused which break point in the list, I only have to take that factor into account when calculating the new slope. There still will be two loops, because it is possible that more than one factor caused a break point at the same frequency.

Why did I include the last part of this example? Because now it seems so simple, but during design it took me a long time. I think, I was probably so fixed on using loops, that I failed to consider whether those loops should examine all elements.

This last paragraph already indicates that design is not a straightforward process, but depends upon search and backtracking by the designer. The next section is another illustration of problems that may arise during design.

§2. Interacting subproblems

If I would look only at the problem of a Bode plot for one factor, I would

*** Actually we should add the amount by which the slope of the asymptote of the individual factor has changed. But since all types of systems which we considered have a horizontal asymptote for low frequencies (if they have a break point), the result is the same.**

say it was clear from the descriptions included in "the expert knowledge", on p. 14-15, that the behavior of Bode plots for single factors is almost completely characterised by a description of their low and high frequency behavior. However, if more than one factor is present in the FTF, these descriptions will have to include a range in which they are valid. As I said before, in a logarithmic plot of log|H| versus log(w) break frequencies are easy points of reference for the behavior of log|H| and allow calculation of the resulting asymptote in various frequency ranges.

A description using break frequencies is easy for terms of the form (1 +jwT) in a FTF. But terms of the form [(jw - a)**2 + b**2] which also occur, cannot be handled so easily. This kind of terms may have an extremum point, see the following figure:

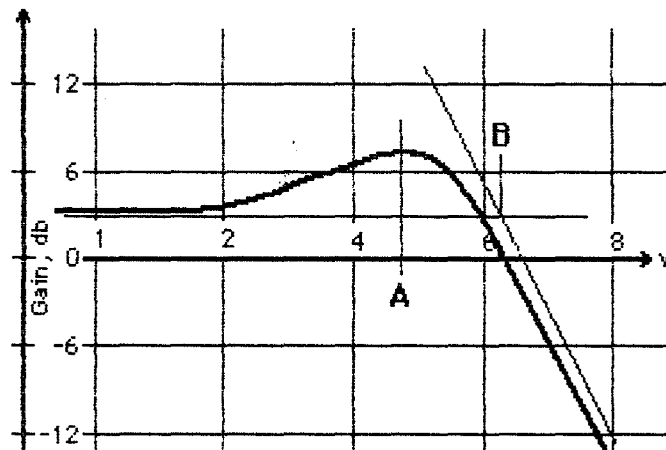


fig. 16 Bode plot with extremum point

If we take as break point the intersection of the asymptotes (point B), near the extremum point it does not seem correct to say that log|H| approaches the horizontal asymptote (the asymptote for low frequencies). If we take point A as break point, the asymptote for high frequencies cannot be calculated very easily (the slope is known, but not the intersection point with the previous asymptote). Since I wanted to make a list of break points for all factors, and use that list to compute the overall behavior in various frequency ranges, I had to make a decision. I decided to take point B as the break point (because that is according to the definition) and kept in mind that the program would have to check between break points whether an extremum point is present.*
The fact that I had to solve this problem, cannot be attributed to the rigid nature of ACTIONS alone (i.e. the fact that advice always must be given in canned sentences, like "Above w = ..."), but is also a problem connected with the nature of Bode plots: something has to be said about the behavior near an extremum point.
The fact that I could not easily take point A as break point has to do with the fact that I have to use those break points in another (sub)task as well. In this way, this problem interacted with the problem in the other subtask.

* Checking for extremum points is not yet implemented in KNOWBODE.

S3. How to translate the flowchart into the SAGE building blocks

Now that I have made sufficiently clear that it is possible to decompose the problem, let us examine how we should translate the flowchart into the SAGE building blocks (AREAS, ACTIONS, RULES, ASSERTIONS, OBJECTS etc.). This is actually not a process which begins when you have a completely finished flowchart, but is done during its design. Not waiting ubtil the flowchart is finished is more or less necessary, because there are limits on the possibilities of SAGE, and also because several choices have different effects on what the user may <u>see</u> when the program is being run. First a few constraints:

1) Loops can only be executed in AREAS or in the external environment. Nested loops in AREAs are only possible under certain conditions (see chapter 10).

2) Output has to be done in ACTIONs.

3) Only RULEs and QUESTIONs can give ASSERTIONs or OBJECTs a value.

Now if you take care of these constraints, you may translate the flowchart into AREAs and ACTIONs. For instance, we try to put every element in the decomposition of the second subtask (which has now become investigation of the low frequency behavior and the break points) into an ACTION. All the ACTIONs together form the AREA <xmine_low_freq\break>. If we should need more ACTIONs to realize a single element in the flowchart, we may consider to form a separate AREA, and just have an ACTION which CONSIDERs that AREA (just as ACTION <Bode_plot> CONSIDERs the subtasks <get_FTF>, <xmine_low_freq\break> etc.). Whether we do this splitting up into AREAs depends on what we want the user to see.

AREAS and ACTIONs form a hierarchy of tasks. But below this are the goal trees. Maybe it is a good idea to say that goal trees should take care of things which the user should not see immediately. An example: the second "action" in what we previously called "get gain and phase for each factor" is "determine type (of factor)". It might be decomposed as:

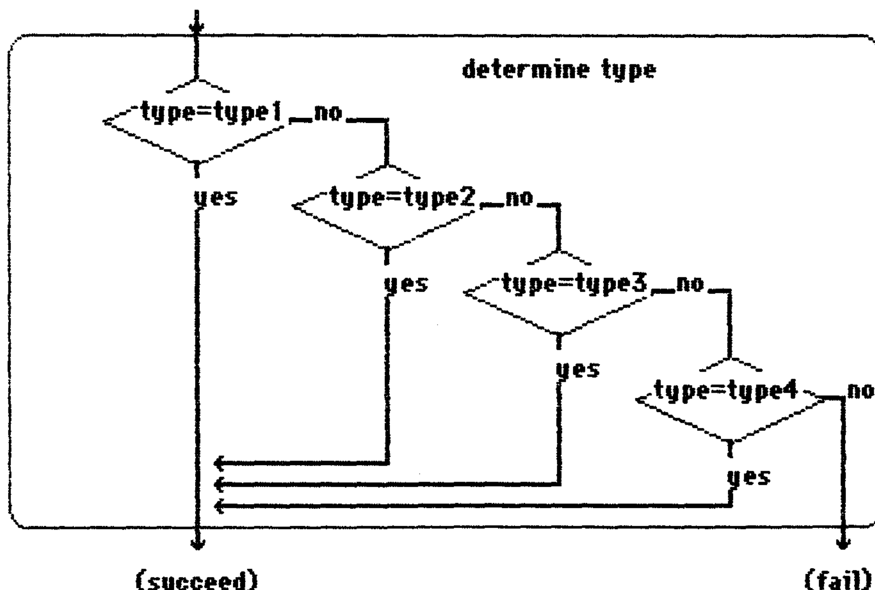


fig. 17 possible decomposition of "determine type"

There are several situations possible in the type of the factor. This may easily be captured in different RULEs, and the user need only see the ACTION <determine_type> (not implemented):

```
ACTION determine_type :
"determines the type of a factor"
        CONSIDER type
```

RULES take care of the *<u>variety</u>* *in situations, ACTIONS of the common parts.*

You may wonder why this ACTION is not implemented. Because it is not necessary! If I examine the break point of a factor (examine whether it is there) naturally, I have to examine what type of factor I am examining. Thus, I moved both the element "get factor from FTF" and "determine type" to the RULE-based part, where they will be called when necessary. In this way, the user will only be confronted (on the ACTION level in <xmine_low_freq\break>) with

```
ACTION add_break_point :
"finding the break point of a factor (if possible) "
"and adding it with the factor to a list"
        CALL add(break_point, cause_factor, list)
        PROVIDED break_point_exists
```

which is in my opinion what is most essential in this AREA. Notice that <determine_type> could have been programmed in ACTIONs, where it would be more explicit, but the designer did not think it necessary.

§4. Brief summary

Let me briefly summarise this chapter.
Design is a cyclic process, where you may
1) try to break up the task into small steps, and make a flowchart.
2) translate (during design) this flowchart into the SAGE building blocks.
Depending on the possibilities of SAGE and the design criteria, you may use
- AREAS as functional entities corresponding with (high-level) (sub)tasks.
- ACTIONS as the "basic" routines in the task, easily visible to the user and which put a stress on the <u>common</u> elements in a task.
- RULES as "basic" routines which are adapted to specific situations.

Chapter 7
AREAs and ACTIONs

Suppose there is a task, which may be divided into several subtasks which can be arranged into a certain sequence. There are several ways of dealing with the investigation of such a sequence*. For instance, we may try to attain the goals as if they are part of a SAGE RULE:

```
RULE execute_task :
"one way to deal with a task having several subtasks"
          you_accomplished_your_task IS TRUE
          PROVIDED you_accomplished_1st_subtask
                   AND you_accomplished_2nd_subtask
                   AND ...(etc.)
```

But what if you fail to attain the first subtask? In the next chapter we shall see that in such a case, in a RULE like this, the other preconditions are not examined, and the next RULE to attain <you_accomplished_ your_task> is investigated. By choosing a structure for the execution of the sequence of subgoals as represented in the RULE above, it is as if the designer says: "I consider the attainment of a subtask so vital, that it's no use going on if you fail in accomplishing it". This indicates that failure is considered a serious event, and that while you're working on some goal you should try hard to reach it. But this "attitude" may be considered a built-in feature of the SAGE system: it is implicitly determined by the procedure that SAGE uses to examine facts, and here used by the designer to execute the sequence of subgoals as desired.

The procedure used to execute the ACTIONs in an AREA is different. Whereas SAGE in goal trees uses a procedure of which one of the characteristics is "I shall examine RULEs to attain a goal until I have evaluated the goal or there are no RULES left", in AREAS it roams about, saying "I shall process all ACTIONs in this AREA, unless I am explicitly told to FORGET it".
This means that *in AREAs, (premature) termination of an investigation must be made explicit, whereas in goal trees this is implicit.*
But also: *Investigation of a goal tree from an AREA will only occur at the time it is explicitly mentioned in an ACTION.*
Should the goal fail to be attained, then it is not possible in SAGE to use the well-known implicit procedure, which people use: when they find they cannot reach a goal, they divert their attention elsewhere and more or less "forget" the goal, but if they notice that the situation has changed, the goal pops back to mind again. Naturally, in SAGE one may try a goal tree later again (actually, not the same but a similar one) but this second attempt has to be explicitly encoded in an ACTION.

* Notice that here we do something of which I said in the previous chapter that it should be implemented in AREAs and ACTIONs. That still remains true if you are considering "major" tasks, which should be "close to the surface", easily visible to the user.

it may not seem clear, but this is one of the basic limitations of SAGE. However, it is possible to remedy it: see Appendix E.

The means which an ACTION has to direct the flow of control explicitly, were already mentioned in Chapter 1: they are "STOP", "RESTART", "REASK", "CONSIDER" and "FORGET".
We will not deal with STOP. It simply halts the execution of the program, and may be resumed by RESUME.
"RESTART"/("REASK") will be considered in Chapter 10, where we talk about multiple use of knowledge in SAGE.
Here we will say only a few words on CONSIDER and FORGET.

CONSIDER *should be regarded as a subroutine call (with constraints).*
FORGET *works only on facts or areas which have been considered, or which are presently under consideration.* It should be regarded as a command to make facts invisible (not unevaluated!) and when applied to AREAS yet under CONSIDERation, as an explicit RETURN statement.*

I shall now demonstrate some features of CONSIDER and FORGET on a small problem: initialization of background memory.

§1. Example: initialization of background memory

As we shall see in chapter 10 we may use the procedure encoded in an AREA several times using RESTART. In order to use results from previous cycles of a loop, an AREA can store some results in background memory. And for keeping track of the number of executed cycles we use increments which are also stored in the external environment.
The buffers where data will be stored or retrieved from during the cycles must be properly initialized, e.g. increments must be set to 1. This is definitely a low-level task, which we want to hide from the user.

We cannot put all the necessary operations in an ACTION and RULES in the AREA itself, because then it would be executed every time the AREA has been restarted. It must be encoded in a seperate AREA.
There are a few possibilities to connect the AREAs necessary for the preparation and the body of the loop:

1] See fig. 18 on the next page. This possibility was used for the AREA <xmine_low_freq\break>. Both this AREA and the AREA <prep_low_freq\break> used for the initialization are CONSIDERED in AREA <control>. When <prep_low_freq\break> is done, it may be forgotten because it is not interesting to the user (only the state of background

*** N.B. when you have an action effect like**

CONSIDER A, B, C

(as for instance in ACTION <Bode_plot>) then AREAS <B> and <C> are not under consideration yet when A is being considered and can therefore not be forgotten!

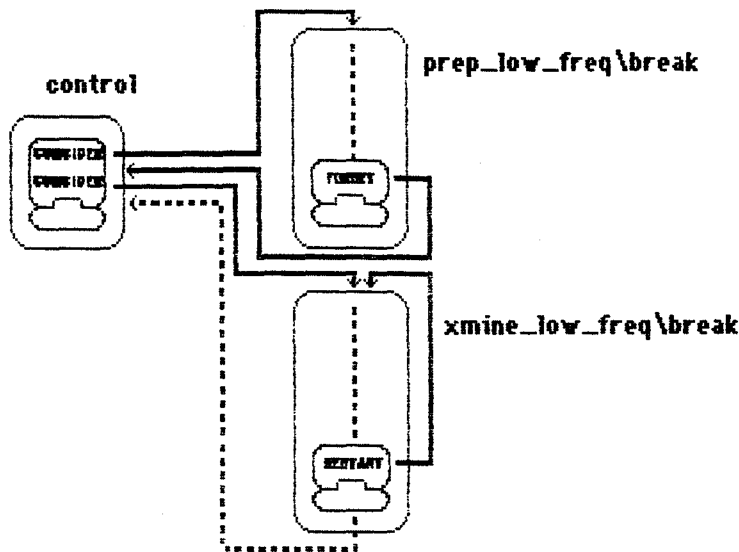


fig. 18 flow of control between areas. Solid lines indicate explicit direction of the flow of control, dashed lines implicit transfer.

memory is altered). Therefore the last ACTION of this area is:

```
ACTION forget_this_area :
"when done, I forget this area, because it is "
"not very important with respect to the actual "
"calculation of the Bode plot"
          FORGET prep_low_freq\break
```

Notice that this ACTION may also be part of <examine_low_freq\break> because the preparation AREA must (naturally) be considered first. But I try to keep "the knowledge" of <prep_low_freq\break> all in one area.
Disadvantages of this method are
1) as far as the user is concerned, the AREA <prep_low_freq\break> and <xmine_low_freq\break> are on the same level of abstraction: because if he looks at LIST Bode_plot (in dialogue 2), both are named as areas to be considered.
2) in an interactive session where the user takes control, he may very well forget to use the area <prep_low_freq\break> and thus not properly initialize the loop.
3) Notice that in dialogue 3 we see that the area <prep_low_freq_break> is entered : "We are considering initialization of..." but we do not see that it ends : there is no message "We have completed considering initialization...". This is due to the use of FORGET.

2] By containing the body of the loop in the execution of the area that initializes it, we give the user less chance of forgetting preparation of background memory. See fig. 19 on the next page.
This possibility was used for AREA <examine_intervals>. Now the AREAs no longer are on the same level of abstraction, no, we pushed the <u>body</u> one level deeper! The AREA which initializes and executes the loop must now have a name and associated text to express that it contains both. As you can see from dialogue 2, the user sees on the highest level only that AREA.
A disadvantage of this method which is not visible in dialogue 3, is: since

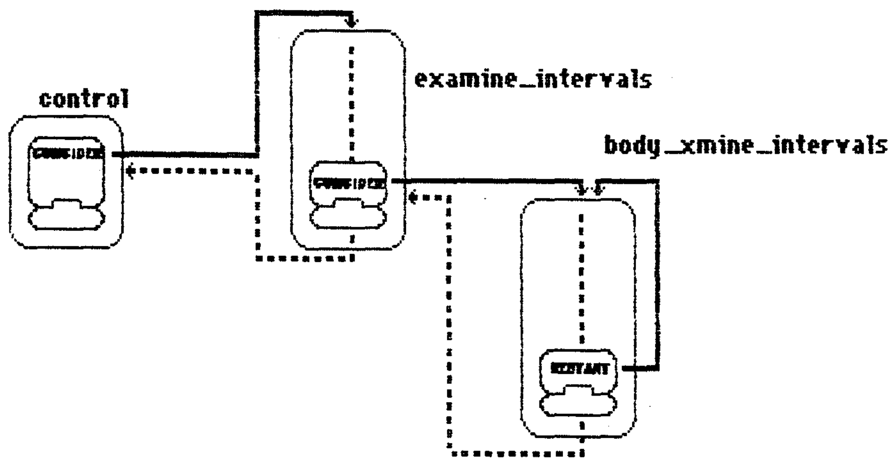


fig. 19 flow of control between areas. Solid lines indicate explicit direction of the flow of control, dashed lines implicit transfer.

the body is now contained in another area, you will see when this area is finished (in TRACE1 or higher levels) something like

> We have completed considering execution of the body of the loop.
>
> We have completed considering initialization of the loop and execution of the body.

Which is a bit confusing because it looks like a strange way to order the areas.

3] A third method, which does not have this disadvantage nor the disadvantages of the first method, but which was probably too obvious to be visible since I didn't discover it until I was writing this report is

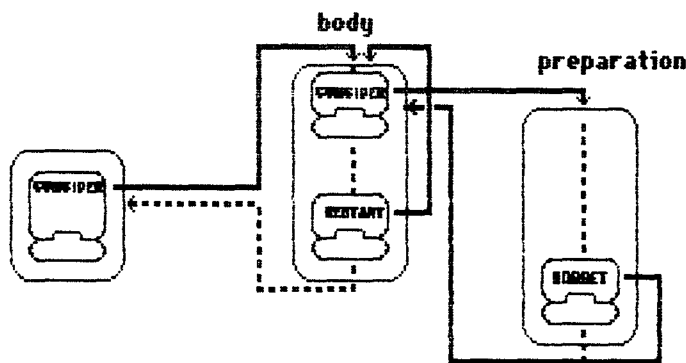


fig. 20 flow of control between areas. Solid lines indicate explicit direction of the flow of control, dashed lines implicit transfer.

This method gives the preparation of background memory its full due: it is a procedure on a lower level then the execution of the body. The method is based on the fact that the ASSERTIONs and OBJECTs in an AREA are examined only once, thus when you CONSIDER the AREA for the second time, no goal tree will be investigated and hence nothing will happen.*

* See first footnote next page.

S2. ACTIONs which should not appear in the flowchart

AREAs consist of a number of ACTIONs. *The ACTIONs in a single AREA should all be "of the same (human) size***", by which I mean that they should deal with the task at hand on the same level of abstraction: in AREA <control> all ACTIONs deal with the task on a <u>subtask</u> level, in AREA <get_FTF>, a subtask, ACTIONs deal with aspects of getting the FTF (like <u>reading</u> it), but they should not deal with aspects as scanning or parsing. It is rather confusing when somebody who is explaining a difficult equation in which among other things multiplications occur, all of a sudden starts explaining how to multiply.

Unfortunately, SAGE is not completely transparent, i.e. I am sometimes forced to use extra ACTIONs in an AREA to realize a certain desired result. I shall now discuss a few cases.

1) ACTIONs to provide interrupt possibilities:

```
ACTION interrupt_process :
"!N!I am about to proceed on the instructions you "
"gave me. As long as you do not answer this "
"question, you can interrupt my line of thought, "
"e.g. for asking ?<name> questions."
            RESTART
            PROVIDED any_questions
```

When the program is running, the only way the user may interrupt it is by waiting for a question. Since KNOWBODE does not have a great deal of QUESTIONs, there are only a few points available where the user may break in.

But the program does make extensive use of RESTART. This means that the program uses certain sections several times, and every time it executes a section again, it will discard all results from the previous cycle.
Which means, that if we want the user to be able to inspect the state of the model, ask questions perhaps, we shall have to provide him with (dummy) questions at appropriate points in order to temporarily halt the program. Such appropriate points are either the first ACTION in an AREA, as in this case, or the ACTION where RESTART is going to occur.
The first ACTION in an AREA might be a good point for the user to change the TRACE level, in order to see the process on a specific factor in more detail.
The last ACTION in a cycle allows him to inspect the evaluated facts before they are returned to the unevaluated state.
Notice how <interrupt_process> creates a small loop to the top of the AREA by using RESTART.

* I am not absolutely sure whether ACTIONs in an AREA which contain no facts, but only ADVISE or CONSIDER <area> effects will not be executed a second time. It is not clear from the manuals (see [22]SAG03 p. 84, for instance). Results of a test I conducted are lost. But also the fact that in LIST AREAS SAGE denotes the status of AREAS as "investigated" does not give much hope of being able to use it again. In the rest of this report, I will take as an assumption that AREAS cannot be reopened.

** or put differently, *"there should be no mixing of levels"*.

2) ACTIONs to store intermediate results:

```
ACTION memorize_part_result :
"I will store some of the results in order to "
"remember them in the next frequency interval"
        CALL put_in_memory([bufferX],
                new_slope_|H|,
                intersect_asym_|H|,
                break_frequency)
```

As noted in 1), programs which use part of an AREA several times with the help of the RESTART command, return all facts evaluated in that AREA to unevaluated before they begin again at the first ACTION. If you will need any results from the cycle just finished in the next cycle, you will have to store those in background memory.

3) ACTIONs providing a more flexible output:

```
ACTION advice_about_slope :
"I give the value of the slope in the above-"
"mentioned advice about the behavior of |H|"
        ADVISE
        slope_begin_|H|,"."
        PROVIDED kind_of_line = NOT horizontal
```

This ACTION is following ACTION <advice_about_|H|>, which gives an advice as indicated in the example on p. 7. The ACTION is necessary, because the advice given is either

[....]represents a horizontal line.

in which case there is no value needed for the slope, or

[....]represents a straight line with slope <slope_begin_|H|>.

where there is an actual value for the slope.
The ACTION is therefore necessary as a result of the rigidity in the output format SAGE provides. See Appendix D, where a similar situation may be found when the program either has to include "1/<variable>" in the output text or not. In that case I could attach a SCALE to <inverse_var> which has the "value" "" (dummy string) when there is no variable.

4) ACTIONs which must RESTART, but not display text.

```
ACTION trivial_factor_2 : ""
        RESTART
        PROVIDED trivial_factor
                AND ( another_factor OR
                (value_factor_cycle > 1.))
```

A side-effect of RESTART is, that the action text of the ACTION which

RESTARTs the AREA, is printed. If I don't want that, as in this case, I am forced to split the ACTION in two ACTIONs, with the same preconditions (and preferably related names), where the last one, which contains the RESTART has only a dummy string "" as action text. Unfortunately this dummy string will still be a dummy string when the user does a LIST ACTIONS or LIST AREA, in which case it is not very clear to the user why there is no action text.

Chapter 8
How to make a goal tree in SAGE

Making a tree is not as easy as it may look.
In a SAGE expert system, statements and numerical entities can be considered as nodes in a (goal) tree. In Chapter 1 (p. 10) a description was given of the rather simple procedure SAGE uses to climb the tree, i.e. to examine facts. But maybe the consequences of this procedure for the designer can be better explained by summarizing the unusual parts in two slogans:

Investigation will lead to evaluation
Evaluated facts will not be reinvestigated*

We shall deal with these statements in the first two sections of this chapter. The last two sections are devoted to the questions in what order the RULEs in your model should be, and how to make an understandable tree. All the examples in this chapter (in fact in the whole report) are on RULEs which equate the value of the conclusion with the resulting value of the rule body (in Appendix A called "equivrules"). Probably no expansion is needed to apply the methods described in this chapter on RULEs where the resulting value depends on several factors ("cumulrules").

S1. Investigation will lead to evaluation

If SAGE starts investigating a fact (as part of an ACTION, or in the course of the investigation of other facts) it will eventually come up with a value for that fact. "SAGE won't take no for an answer"**.
This means, that in a program you should avoid a fact being investigated as long as it is unknown whether useful evaluation is possible. An example:

```
RULE calc_break_point_2:
"....................................."***
        break_point IS square_root(alpha*alpha +
                          beta*beta)
        PROVIDED type(factor) = second_order
```

Had the precondition ("type(factor) = second_order") not been present and investigated before the execution of the rule body, the objects <alpha> and <beta> would have to be investigated by SAGE. However, these objects only have significance for second order systems, so SAGE would probably find their values to be "unknown" (worse, since I happen to know that they get their values simply by retrieving a value from the external environment:

* except through RESTART/REASK, or explicit use of commands by the user. See "multiple use of knowledge".
** In fact, SAGE will. But it won't gladly take "unknown" for an answer, and in that case will keep on trying other RULEs for a better answer.
*** see p. 9 for the associated strings.

```
RULE obtain_alpha :
"gets the value alpha of the factor we're discussing "
        alpha IS get_alpha(factor)
```

and similarly for <beta>, SAGE will probably come up with some meaningless value). It is clear that such a investigation should have been avoided.
Here we have seen a first means of preventing the investigation: preconditions. *Preconditions can prevent investigation of the (RULE/ACTION/QUESTION) body*. They can postpone, but not prevent the evaluation of the conlusion of a rule: if the precondition fails, <breakpoint> will still get a value, but not by this rule.

Another situation where it is very useful to prevent investigation of a (rule) body is if the (rule) body contains functions which have effects on the outside world:

```
RULE GRASP_block_Z :*
"grasps block Z"
        I_grasped_block_Z IS
                grasp(block_Z)
        PROVIDED Z_has_a_clear_top
                AND my_hand_is_empty
                AND my_hand_is_over_Z
```

The FUNCTION <grasp>, which results in the actual grasping, should not be activated unless all preconditions are met.

We may use this example to illustrate a second means of preventing investigation: operators like "AND" and "OR". Unlike in "ordinary" logic, we can say that these operators have <u>several</u> functions:

[1] They stand for their "logical" definitions (adapted to the fuzzy logic which can be employed in SAGE, see [22] SAG03 p.65):

<X> AND <Y> = min(<X>, <Y>)
<X> OR <Y> = max(<X>, <Y>)

[2] They determine the order of investigation: (see example) <Z_has_a_clear_top> is investigated before the assertion <my_hand_is_empty>.

[3] They can prevent the investigation of facts ([22]|SAG03 p.81):
If "<X> AND <Y>" is encountered and either factor is already evaluated to FALSE, then the other factor will not be investigated.
If "<X> OR <Y>" is encountered and either factor is already evaluated to TRUE, then the other factor will not be investigated.

So, in our example, if <my_hand_is_empty> turns out to be FALSE (for

* Example taken from "MOVER" a program in SAGE made by the author, to move blocks.

some reason or other the robot couldn't get his hands free*) then the next assertion, <my_hand_is_over_Z>, is not investigated. This is fortunate, because accidents might happen if the hand is moved over to Z still holding some, possibly large, object.

In short, *operators like "AND" and "OR" may prevent investigation of facts following the operator.*

The built-in function "KNOWN" is a third means of preventing investigation in SAGE. We will illustrate it in an example which uses also the previous means:

```
ACTION help_Bode_plot :
"I set some of the features which will be used "
"during the calculation of a Bode plot, if wanted"
        CONSIDER xtra_xplanation_want
        ALSO
        CONSIDER interrupts_possible
        PROVIDED KNOWN(Bodeplot_wanted)
                    AND Bodeplot_wanted
```

This action is part of the AREA <user_wishes>. This AREA, which tries to determine what amount of advice the user wants or needs, is entered as a result of the first action effect in ACTION <Bode_ploT> from AREA <control>:

```
ACTION Bode_plot :
"I will now calculate a Bode plot"
        CONSIDER user_wishes
        ALSO
        CONSIDER get_FTF, prep_low_freq\break,
                 xmine_low_freq\break,
                 examine_intervals
        PROVIDED Bode_plot_wanted
```

But suppose that a future version of KNOWBODE also allows <u>polar plots</u> to be made. For such a version, the existing AREAs <get_FTF> and <user_wishes> need little change: perhaps a few extra ACTIONs to take care of specific user wishes associated with polar plots. The actual (overall) calculation of the polar plot may be directed by an ACTION very similar to <Bode_plot>. Suppose such an ACTION (i.e. <polar_plot>) was positioned in <control> <u>in front of</u> <Bode_plot>.
Now if a user wants a polar plot, then as a first action effect the flow of control of the program will be directed to AREA <user_wishes>. In this area, the program will encounter <help_Bode_plot>. Had its precondition been:

```
PROVIDED Bode_plot_wanted
```

the program would reason as follows: "here we have an ACTION with as a

** We don't just <u>investigate</u> <my_hand_is_empty>, we want it to be TRUE. See the next paragraph: this is an example of the slogan "You don't want to come back empty-handed to the top of the (goal) tree".

precondition a yet uninvestigated* ASSERTION. Let's examine it". Meaning, that the user having just answered "yes, I want to see a polar plot", all of a sudden is confronted with a question whether he wants to see a Bode plot! As a designer I chose to postpone that question until the (second) ACTION in <control> is investigated, and so I used the function KNOWN.** *KNOWN may test whether a fact has been evaluated without causing its investigation (and subsequent evaluation).* If <Bode_plot_wanted> has been evaluated before, KNOWN will return TRUE and then we can check what value it has (true or false) in the second part of the precondition of <help_Bode_plot>. If it has not yet been evaluated, KNOWN will be FALSE, and thanks to the AND operator (previous method!), no further investigation will ensue.***

S2. Evaluated facts will not be reinvestigated

This has a very positive effect, especially on consultational programs: SAGE will investigate a fact only once and then remember the value, so the user will not be confronted with the same questions over and over.

But there are also negative effects. If a fact has been given a value which you don't like, it will cost you extra work to improve the situation. See for an example Appendix C on <read_FTF>. This means that *in designing a program you should avoid a fact being evaluated (getting a value) as long as the (expected) value is not what you want.* Or as a slogan:

> **"When you come back to the top of the tree, you don't want to come empty-handed."**

Illustrated by the example a few pages back (GRASP_block_Z): when investigating <Z_has_a_clear_top>, we don't want to see the current value, we try to make sure that it's TRUE. Here we have a real goal, not just an unexamined fact.
In programs both real goals and unexamined facts will appear. Look sharply what you want.

What means does a programmer have to effect this avoidance of evaluation?
We have already seen, that *preconditions on a RULE or a QUESTION can prevent that some fact (conclusion) will get a value by that RULE or QUESTION.* SAGE will then try another rule or question to give that fact a value.
A similar effect can be obtained by having the RULE- or QUESTION body evaluate to UNKNOWN. As long as other means of establishing a value for that fact are available, SAGE will treat an UNKNOWN fact as "not yet

*** Because in a normal course of the program this assertion will be investigated as a precondition of ACTION <Bode_plot>, which has a position after <polar_plot> in AREA <control>, and is therefore not yet under investigation.**
**** A possibly better solution is to use an option question like <ask_interrupts>, which can give values to several facts at the same time!**
***** Notice that people use the word "known" usually in the sense "known to be true". SPL might think of preventing confusion by changing the name to "EVALUATED".**

evaluated" and will go on investigating it.
A RULE- or QUESTION body which evaluates to UNKNOWN, transfers evaluation of the (conclusion) fact to other RULES or QUESTIONS. See also Appendix C for an example.

§3. The order of applicable RULES

In what order should the set of RULES which may establish a value for the same fact be put into the model's source listing?

A first approach, which is very well applicable in the KNOWBODE case, is to use no particular order. It is as if the designer thinks: "I know that SAGE will try all applicable rules until it finds an answer. If there is more than one "good" answer for a fact, I don't care which one he takes. But by using preconditions on the rules I shall limit the situations in which they can come up with a "good" answer". This approach makes it very simple to add new RULES to the model.

But it is not always desirable to use this approach. Consider for instance the case from a few pages back, where there was a RULE to establish the goal "I have grasped block Z"(<I_grasped_block_Z>). In the same model there is another RULE to establish that goal:

```
RULE obvious_GRASP_Z :
"when I am holding block Z, it is"
" pretty obvious I have grasped Z."
        I_grasped_block_Z IS TRUE
        PROVIDED I am holding(block_Z)
```

Most people would probably agree that the program is downright stupid if it doesn't test this rule before rule <GRASP_block_Z>. It saves a lot of useless effort (attaining all goals in the preconditions and rule body of <GRASP_block_Z>) on the side of the program, so why not make use of the fact that the order in which SAGE examines RULES is fixed, and put this one in front of the other?
Since I was talking about useless effort on the side of the program: what makes me so sure that evaluating the FUNCTION <holding> costs considerably less effort than evaluating the preconditions and rule body of <GRASP_block_Z>? It may be that <holding>, which represents a subprogram in the external environment, involves the execution of thousands of FORTRAN statements, and not so little effort after all.
Be warned. Every RULE to establish a fact connects that fact to a tree of other facts. *If you have knowledge on the effort it costs to investigate some tree, or on the importance of the situation in which a certain tree is applicable, use that knowledge while designing the program.* "Effort" naturally also includes effort on the part of the user: if in a consultational program a fact may be established by two RULES, of which one involves asking the user one simple question, and the other involves asking a lot of difficult ones, which RULE should the program try first?

This knowledge on what is "simple" and "difficult" is part of the knowledge the expert on the problem should provide.

Concerning the order of RULES, I'd like to discuss two more "slogans" which are the implied result of the procedure SAGE uses to investigate facts. The first one is: *investigating a RULE/QUESTION for some fact, implicitly means that preceding RULES/QUESTIONS have failed!*
Consider the following two RULES from KNOWBODE:

```
RULE calc_prev_point :
"if this is not the first break point, we can "
"take the value of the previous break point "
        previous_point IS recorded(prev_break_freq)
        PROVIDED NOT first_break_freq

RULE calc_prev_point2 :
"if there is no previous break point, we can take "
"w = 1. as a frequency for which we know the "
"value of the asymptote of |H|"
        previous_point IS  1.
        PROVIDED first_break_freq
```

Since <recorded> is a simple FUNCTION to retrieve a value from background memory, and <prev_break_freq> merely indicates the position in this memory, we know that the second rule is activated (only) when we are dealing with the first break frequency (<first_break_freq> = TRUE). We may leave this second rule out and make 1. simply the DEFAULT value of <previous_point>. In this case, it is perfectly valid, but in general I would suggest against using that kind of knowledge, because it is not explicit and may therefore be overlooked by future designers who wish to add rules.

The second, related, slogan is: *investigating some fact implicitly means that there is a situation which justifies this investigation!**
In the first section of this chapter we discussed what would happen if the precondition "type(factor) = second_order" was not present in RULE <calc_break_point_2>. It would mean that RULE <obtain_alpha> could be activated in situations for which it is useless, i.e. when there is no <alpha> for the factor under investigation. We can also state that otherwise: since the precondition is there, we know that when the RULE <obtain_alpha> is activated, it is activated in a context that there is an <alpha> present. This justifies the fact that the same precondition was not added to rule <obtain_alpha>.

*** You might try to encode "The order of looking for things" ([8] p.39) into SAGE RULES**

§4. How to make an understandable tree

It is good to realize that "understandable" means "understandable for the user", because SAGE will 'understand' anything as long as it is syntactically correct.
As said before, KNOWBODE is not a perfect expert system. The suggestions given below are certainly not followed precisely in the model. But during design one should reach for performance rather than for <u>easy</u> understandability. "Understandable" will do. Also, if any of the suggestions given below may seem unnatural to a designer in SAGE who reads this, I advise him/her <u>not</u> to feel obliged to adopt my conventions but to use his or her own, because those are the ones that will be used most easily.
The exact text associated with the elements in the tree has to be adapted through series of consultations with an actual end-user. This is also "design by prototyping".

In KNOWBODE, the program refers to the user as "you", to itself as "I" and (in the standard SAGE sentence fragments) to the parties involved in the dialogue as "we".

<u>§4.1. ASSERTIONS</u>

ASSERTIONS should describe relations between objects (in reality).
The name should be the shortest expression of the statement made in the assertion. The first level of associated text should present the statement, preferably very short. Extra information can be buried in deeper levels. *Don't forget that the user will encounter this assertion as part of a whole tree:* when going through the tree (with FACTORS or WHY command) he doesn't want to be flooded with the same information over and over again.
SCALES attached to ASSERTIONS are useful for output in ACTIONs, but on lower levels (i.e. in TRACE 2 or 3) they may create confusion. An example:

```
ASSERTION behavior_|H| :
"there is no break point, log|H| vs log(w) is a "
"straight line everywhere"
          DEFAULT FALSE
          USING range_scale
```

1) the name <behavior_|H|> does not express the statement. However if it had done so, it would have created confusion in the listing of ACTION <advice_about_|H|> (see p. 7).
2) the expression of the statement is correct.
3) Use of the SCALE <range_scale> provides flexible output in ACTION <advice_about_|H|>. But on TRACE 3 level the user may encounter things like:

> Result for behavior_|H| :
> The likelihood that there is no break point, log|H| vs log(w) is a straight line everywhere is For low frequencies, the asymptote of |H| in a log|H| vs log(w) plot (-100.00)

§4.2. OBJECTS

For OBJECTS, almost the same recommendations apply as for ASSERTIONS. OBJECTS do not express relations, but denote entities. The first associated level of text should be a (short) noun phrase (whereas with ASSERTIONS it is a sentence).
SCALES may produce both readable output and better understandable traces at deeper levels.

A good example of an OBJECT is <list>, with three levels of associated text (see dialogue 3) and an associated SCALE <memory_scale>, which in TRACE 3 will display as

Node <list> evaluated to position LIST-1 (81.00)

Another example is <break_point>:

```
OBJECT break_point :
"the break frequency"
MORE
"the frequency where one region of asymptotic "
"behavior of gain |H| and phase arg(H) meets "
"another"
          ( 0.0, maximum_real)
```

which has a neat division between text in the first level and deeper levels, but which lacks a clear definition on how to compute the break point ("it is defined as the frequency where one asymptote intersects another").

§4.3. RULES

RULES are the means by which a fact may obtain a value.
In SAGE a RULE can only establish a value for one fact (although during the process of their execution other facts may be evaluated). Therefore the name of the RULE should express for which fact the RULE is meant, which is useful e.g. if you LIST all RULES in a certain AREA, or it should express in what respect this rule is different from other rules to establish the same fact. This is useful when you ask for the FACTORS which may establish some fact. See dialogue 3: <calc_coeff_1st_term1> etc. display in their name only for which fact they are meant. There are no good examples in KNOWBODE of the other kind of names, but RULE <judge_by_credibility> from the Appendix in part 1 seems to express what I mean.
In the names I used "get" or "obtain" to indicate simple retrieval of values, "calc" to indicate that some kind of "calculation" was involved, whereas "check" is more an indication that we are trying to confirm some assertion.
In the first associated level of text of the RULE the procedure encoded in this RULE should be described. However, I don't think you have to repeat everything that you see in the rule body in this text, if you want to keep it short. This means that I think the text should support the rule body. An

example of this may also be found in dialogue 3: RULE <calc_rel_position> :"because this product of ...".
But if you use LIST RULES you will not see the rule body and be confused if the associated text does not contain a description of the actual calculation. Well, you can't win them all. I think FACTORS RULE + a short description is more important than LIST RULE.
Deeper levels of associated text are syntactically possible, but I never succeeded in making them visible in an actual session. I don't know why.

Another aspect which I would like to stress in RULES is, that because the actual procedure which calculates the conclusion in the RULE is a whole tree of RULES rather than the topmost branch (RULE) which I called "the" procedure, you should "keep your knowledge local" and "there should be no mixing of levels". This is already implicit in the previous section and chapter 7, but I will discuss it here in the light of understandability.
When there are RULEs in a model of a type like this:

```
A IS ...
PROVIDED B
         AND C
         AND D
```

i.e. RULES with more than one precondition (and there are other RULES to obtain B, C and D), it is possible to move the preconditions between rules like this:

```
A IS ...
PROVIDED D

D IS ...
PROVIDED B
         AND C
```

(there are a few other ways possible). This reminds me of the remark in the previous section: investigating some fact implicitly means there is a situation which justifies this investigation. But you should be very careful to see if you are doing the right thing:
1) the RULE to attain <D> has been specialized to a case where <B> and <C> are preconditions. Are there any situations in which you need <D> but where the goals <B> and <C> cannot be attained? In that case the second way of encoding the knowledge into RULES forces you to use an extra RULE.
2) In the second version preconditions <B> and <C> have been moved to deeper levels and on the level of the first RULE they are implicit.
Consider for instance a new (not implemented) version of RULE <GRASP_block_Z> (see p. 63), where <Z_has_a_clear_top> and <my_hand_is_over_Z> have been added as preconditions to the RULE to obtain <my_hand_is_over_Z>:

```
RULE GRASP_block_Z :
"grasps block Z"
         I_grasped_block_Z IS
                  grasp(block_Z)
         PROVIDED my_hand_is_over_Z
```

Now it is taken as implicit that <my_hand_is_empty> is true etc. The user may find this in the deeper levels, but how deep does he want to dig and what does he understand from the top level?
The converse may also happen, when you add preconditions from the rule to attain <B> in the first rule, before precondition <B>, which may give you too much detail visible in the first RULE. What you should do is "keep your knowledge local" where it is needed, in chunks of about the same "human" size. A bit vague, but you'll manage.

§4.4. QUESTIONS

The name associated with the question should express for which fact the question is intended, or the specific nature of the QUESTION (see RULES; this is a similar case). I used "ask" as in <ask_Bode_plot> to indicate for which fact the question is intended.
The associated text of the QUESTION should naturally be the question that needs to be asked to give a certain fact a value. Deeper levels of associated text can elaborate or rephrase the question.

Although a human-like outlook of the question answering behavior may tempt the user to overestimate the program and consequently leads to disappointment ([6]) I think it is the best way yet to use in questions, because I rather have a disappointed user of an expert system than a confused one.
A neutral question like

Is the frequency-transfer function available?

(currently in use in KNOWBODE) proved to be not as clear as

Can you give me the frequency-transfer function?

But another version, which still directly addresses the user

Can you give the program the frequency-transfer function?

introduces (in my opinion) a mysterious third party: there is you (the user), there is the program, and then there is a "thing" feeding the program. (Although this does shed some light on the parties involved in the standard SAGE sentence fragment "we are considering...).
If you don't know which approach to choose, put the alternatives in deeper levels of text associated with the QUESTION.

§4.5. FUNCTIONS

In the next chapter (section "grey zone") I shall elaborate on the name and associated text of FUNCTIONs. For the moment it suffices to say that although it is syntactically correct to associate more levels of text with FUNCTIONS, I never managed to make them visible in a consultation session.

Chapter 9
Design of the Interface and External Environment

In chapter 1 I mentioned that there is a possibility to connect SAGE to other computer programs.
These programs may be used to supplement the features of SAGE, e.g. provide it with access (other than by terminal) to the world outside the computer, by programs to handle robot hands and eyes.
We will call the user defined system containing all these programs "the External Environment". SAGE can have access to this environment through FUNCTIONS and PROCEDURES. A CALL to a procedure in an ACTION, or the investigation of some function in an expression, will cause the value of the arguments to be transferred to a specific external program. The arguments, i.e. ASSERTIONs, OBJECTs or CONSTANTs will be evaluated by SAGE before the actual call is made.
FUNCTIONs return a numerical value from the external environment, or return UNKNOWN.

SAGE can identify which part of the external environment to use by a system key, a numerical value attached (by the designer) to each function or procedure in the introductory block of a model. In the rest of the model, the designer may call functions/procedures by name, but SAGE will use the value to call the right spot in a program named XXSWIF ("SWItch to Xternal", the "F" denotes that it is a FORTRAN program, in our case). From then on, it is up to the external program, until it returns to SAGE. See [24]SAG04 and part 3 of this report.

From the analysis in chapter 4 of the external environment used for KNOWBODE, we may derive that here the extensions of the possibilities of SAGE are:

1) Allowing the user to enter several data elements in one chunk (the FTF), in the function <read>. See Appendix C.
2) providing a memory function for AREAs which are RESTARTed. In this memory, intermediate results of calculations and increments are kept.
3) providing a much simpler means of carrying out simple sort & search tasks than SAGE does. An example: the function <sort>. See Chapter 10 "Multiple use of knowledge".
4) providing a possibility to use loops in output routines, which can easily be used time and again at different spots in the model: see procedure <value_|H|> (Appendix D).
5) a (future) possibility of carrying out calculations on several (semantically related) pieces of data at the same time. See Appendix D.

These extended possibilities are the result of the environment designed. The need for these extensions came from the task at hand and the given features of SAGE: a task like making a Bode plot from an FTF consisting of several factors seems to elicit the use of loops , sequences of

calculations which are repeated several times and which possibly use the results of previous cycles. The extensions 2), 3) and 4) just mentioned are all related to this multiple use of knowledge.
I would therefore prefer not to speak of the extension of possibilities of SAGE, but rather of using the external environment to ease constraints. Comparing programs with SAGE, one shouldn't forget that although these external programs win on some points, they loose on others.

In order to give a more general idea of how to use the External Environment when designing an expert system in SAGE, I think it will be better to leave the functional description and consider (paradoxically*) an historical account of the design.

§1. Historical account of part of the design

Naturally, when designing KNOWBODE I did not start on the external environment, but on the model.
Originally, I intended the model to obtain the data on the frequency-transfer function (i.e. AREA <get_FTF>) by asking a series of questions from the user (see Appendix C for an example). Soon I was confronted with problems which were related to the facts that an FTF consists of a variable number of factors which may be of several types, and may easily contain two or more factors of the same type.
In the flowchart I devised of the problem I simply created a loop: KNOWBODE should use part of its questions and facts again. This proved to be not very easy. This, and the fact that it would take the user quite a long time to enter an FTF by answering questions, made me decide to use a FUNCTION (or PROCEDURE) to read the FTF.

Such a function will load data into the external environment. This means that this data must be stored there until the model needs it and takes it out. This "must be" is not true: external programs may also "offer the data at their own initiative"- but not in SAGE, because SAGE is a backwards chaining system , i.e. a system which hypothesizes a conclusion and uses rules to work backward toward the hypothesis-supporting facts (Winston, [19] p.185).
If I let the data stay in background memory, making multiple use of the procedures (AREAs and RULE trees) just means functions must pick up their data from different positions in memory (see Chapter 8).

Having decided to put the data in background memory, I began thinking how I should represent this data in there. There are several strategies to represent rational polynomial expressions like the FTF is in a computer (see [11] Pavelle p. 110). Choosing a suitable representation depends critically on what you want to do with the data in it. For instance,

*** Paradoxically, because it would seem that if you want to know how you may use the External Environment, it is better to consider what it does than to consider how you made it. However, since the constraints on the problem were not easily visible at the beginning, and a lot of different implementations of the external environment and its interface are possible, and these are conditions which are probably very common in design, it seems worthwile to examine why this particular configuration was chosen.**

representing a first order system like "(1 + jw 3)" in this way:

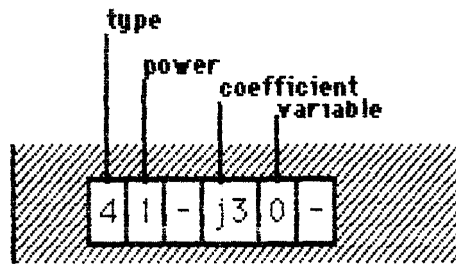


fig. 21 possible implementation in background memory

enables functions like <type> (retrieving the type of a factor) to be very simple, whereas

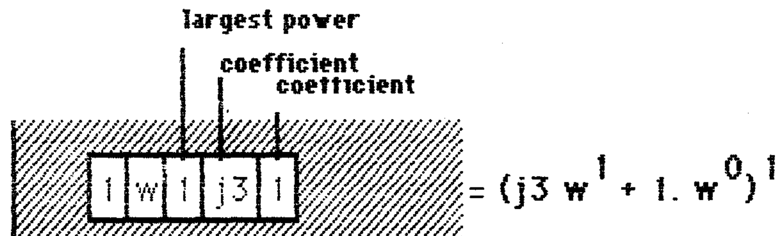


fig. 22 possible implementation in background memory

is easier for functions which add polynomials. At the time I was designing the system it was unclear which functions were to be used, and since I did not want to choose a representation which would have to be altered completely in the future (flexibility criterion of design!) I decided to wait and work on the model.

I shall now dicuss the design of two different FUNCTIONs, <type> and <current_element>, which are repesentative for the majority of interface routines.

At some point or other during design of the model I needed the <u>type</u> of the factor the rules were investigating, e.g. in a RULE like

```
RULE check_break_point :
"a break point will occur if the factor(partial "
"system) being investigated is type 3 or 4, "
"that is, a first-order or a second-order system"
          break_point_exists IS TRUE
          PROVIDED (type(factor) = first_order)
                    OR (type(factor) = second_order)
```

At first it may seem not clear why the tree structure ends in a function <type> here: the FTF the user types in is examined by a scanner and a parser program, which determine the type of a factor in terms like "if the first symbol is a left parenthesis '(', and the second symbol an integer equal 1, and the third... ...then the type of this factor is type 4"*. But

* Notice that I use an IF-THEN structure to describe this procedure! N.B. the parser used in the external environment of KNOWBODE, is <u>not</u> based on such rules.

people need only to be told "factors of the form (1+ jw <x><T>), where <x> is ... ..., i.e. first order systems, are called type 4 systems" as is done in several rules in the model.
In the previous chapter I already mentioned that the point where we stop putting knowledge in facts and rules (and continue in normal programming languages) depends on the performance the program must be able to reach, and on what we want to be visible to the user. There is no clear dividing line, because designers are another kind of users than end-users of an expert system, and have different wishes on what should be visible, but in this case the balance swung in favor of the end-users.
Notice that, once I have decided to use a function <type> to retrieve the type of a factor from background memory, I can go on designing the model (in fact, even running it to check simple explanation capabilities) without having to know how this function should be implemented! *
Somewhere in the future I will have to come up with a program which can perform that function.

In a similar way, when during design I needed the current element from a list of break points, I figured that this would be easier to implement in a "normal" computer program than to go on in RULES for a few levels, and I decided that some FUNCTION should come up with that value:

```
RULE get_break_frequency :
"we take an interesting frequency from our list "
"of interesting frequencies"
        break_frequency IS current_element(list,
                              break_point_cycle)
```

<break_point_cycle> was a CONSTANT, indicating a position in background memory where the number of break frequencies examined is stored.
According to my opinion (as expressed in dialogue 3) the user also is not interested in how you take this element out of the list, but more in the process as a result of which it was put into the list.
Looking at this function, the reader may wonder why the result desired in the construction, which is actually

```
break_frequency IS current_element_list()
```

is not generalized to

```
break_frequency IS element(current_position, list)
```

where <current_position> is an OBJECT which directly holds the number of break frequencies examined, instead of just indicating a position where this value is stored, like <break_factor_cycle>.
Naturally, I generalized <current_element_list> to <current_element>, to be able to use the same function on possible other lists, like

* But, as we will presently discuss, the design may have effects on the implementation needed.

<cause_break_point>, a list of all factors causing a certain break point. It seems such a waste to use a program in the external environment only once. But if I used an OBJECT <current_position> to keep track of the desired element of the list, I would have to introduce also a RULE like this:

current_position IS recorded(break_point_cycle)

I would still have to store the value in background memory, because in order to change it, it will have to be returned to unevaluated by RESTART and will then be forgotten by the model (see Chapter 8). But I think that this RULE is hardly interesting to the user, and should therefore better be included in the function itself.
This does not explain why <break_point_cycle> is present among the arguments of <current_element>. If I let the function take care of the value of the current position in the list, why not also let it take care of where that value is stored?
If the function is applied to a list other than <list>, the function should be able to recognize that this is a different list, and keep track of the current element for that list. This is rather easy to do if you know the possible lists in advance. But if we go on to consider how the function will do it, we see a problem: the function will probably keep different increments in store for all its possible arguments. But where are these increments initialized? When do they need to be initialized? How can the function do that (in advance of the actual function call)? This depends on the specific argument lists concerned and the structure of the model.

To deal with this problem in an easy way I added the extra argument, visible to the designer, as a reminder "this is the location where the function keeps track of the current element in the list named in the first argument. I (the designer) must take care that this location contains the right value".

Notice that our desire to generalize the function (i.e. not making <current_element_list> but <current_element>) now forces us to be more specific on the function call! And notice that this was the result of a "bottom-up" process: it was found when I was thinking of how the function could be implemented in the external environment.

By the way, why didn't I generalize the function <type>? It is easy to make a function and rules like

type_of_factor IS get_from_array(factor, type_element)

We can then use this same function in

variable IS get_from_array(factor, variable_element)

But these calls which look very similar on the model side, are only similar on the external side if type_element and variable_element are quantities actually stored in a list! See the pictures a few pages back: both types of background memory can be called by these functions, but only in the first

kind can a single type of routines handle both calls.

Here we see that generalizing functions on the model side imposes restrictions on the structure and functions (if we want to keep them simple) in the external environment. But the converse is also true: if we want to have a simple kind of background memory, e.g. a single real array, and allow only a few generalized FUNCTIONS, as in

```
RULE get_value :
"retrieves a value from some memory position"
        value IS recorded(position)

RULE do_store_a_value :
"you store a value somewhere in memory"
        store_a_value IS  put_on_record(position, value)
```

(these RULEs are taken from AREA <control>, but are not used in the program). As I mentioned earlier, a function like <current_element> can be realized using only these FUNCTIONs, but it will take a number of RULES. The functioning of these rules is only interesting to the designer, and probably it could have been done easier in a program on the external side.

Let me summarise this proces of design of the interface and external environment.

When, during design of the model, I decide I need a FUNCTION or PROCEDURE (i.e. access to and use of the External Environment), I have a wish in a specific situation. (As said before, the need to use the external environment is caused by the wish to ease the constraints on SAGE).
Instead of a FUNCTION/PROCEDURE fit for the specific situation, I can use a more general one (using arguments in the function or procedure call), thereby aiming at easier implementation on the external side. (see fig. 23)

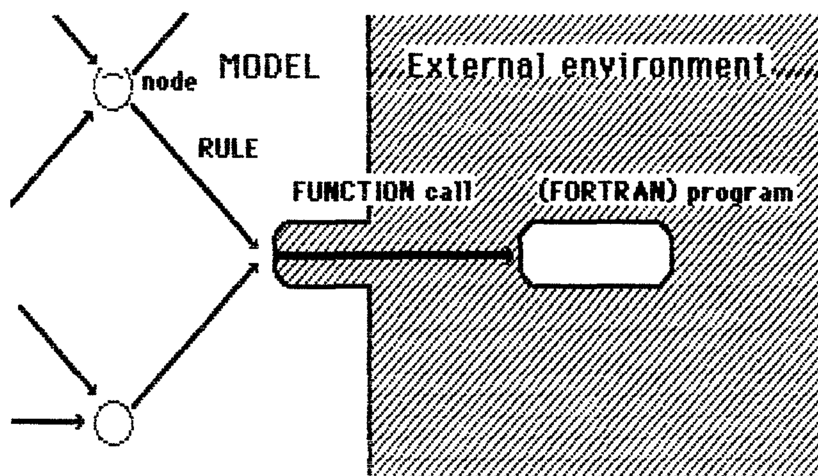


fig.23 FUNCTION calls using the same FORTRAN program

However, if you generalise on the model side, without looking at possible implementations, it is possible that an extra level of programs will have to be introduced at the external side. (see fig. 24).

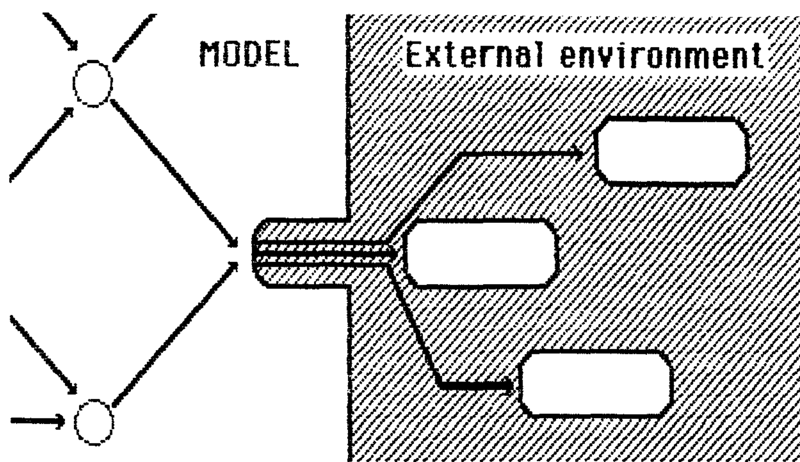


fig. 24 same FUNCTION calls use different programs

If there is an actual inplementation on the external side, it may determine whether it is useful to generalize function calls on the model side. (see fig. 25)

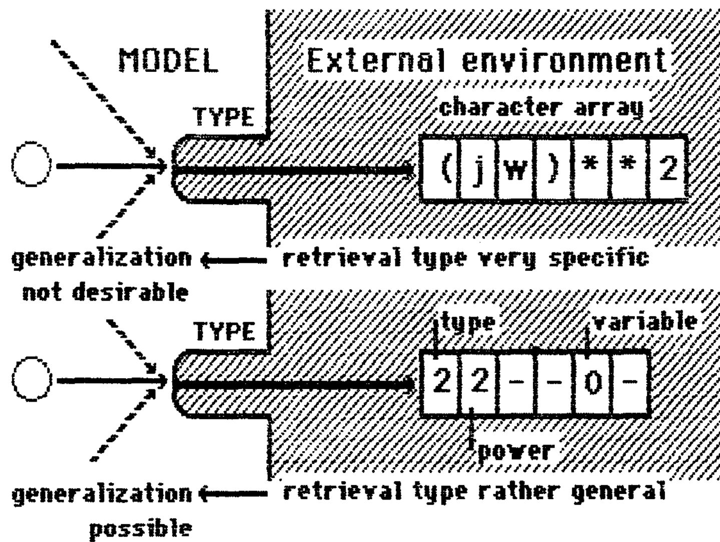


fig. 25 different implementations

S2. grey zone

In view of <u>flexibility</u>* and <u>easy visibility</u> for the designer, there will always be (partial) goal trees on the model side, which must realize the specific wishes in terms of the available FUNCTIONs. This is what I called the "grey zone" in Chapter 4.
When the model is (almost) ready to run, we know the specific situations in which functions are called, and we may try to restore the <u>specificness</u> of the call. This is necessary, as you may see from the following example: in KNOWBODE there is an ACTION <clean_and_calc_2>:

```
ACTION clean_and_calc_2 :
"clean the list necessary in background memory"
        CONSIDER I_cleaned_list
        PROVIDED I did put_on_record(cycle_3, 1.0)
```

* If you have the right basic functions, design is easier on the model side than on the external side.

Now if the user runs the model in TRACE 2 or 3 mode, he may be confronted with a message like:

"The goal we are considering is: I put the given value (2nd argument) in the indicated position in background memory (1st argument)"

You'll probably agree that this is not very understandable. Here we see the "grey zone" quite close to the surface. What I should have done here was make the specificness of the situation visible. There are a few means to do that:

1) Change the name and associated text of the FUNCTION. This is a good idea, because it has no effects as far as extra external programs, extra RULEs etc.

However, this is only applicable if the text associated with the FUNCTION may be used in all call situations. But notice from Chapter 4, that <current_element> and <next_factor> use the same external routine! So it is possible to make use of it.

2) Change all CONSTANTs in the arguments to OBJECTs. In this way, an extra layer of associated text may be introduced. <break_point_cycle> is such an OBJECT, with only a CONSTANT as its DEFAULT value.

3) I may also introduce an extra layer of of RULEs: instead of using the FUNCTIONs directly in ACTIONs and (some) RULEs, I put a RULE in between, like in

variable IS get_variable(factor)

This allows me a specific explanatory text with <variable>, and other features, like use of the SET command (see dialogue 4). A disadvantage is, that if a FUNCTION is used in more than one AREA which are possibly RESTARTed, I have to introduce an OBJECT+ RULE for each AREA.

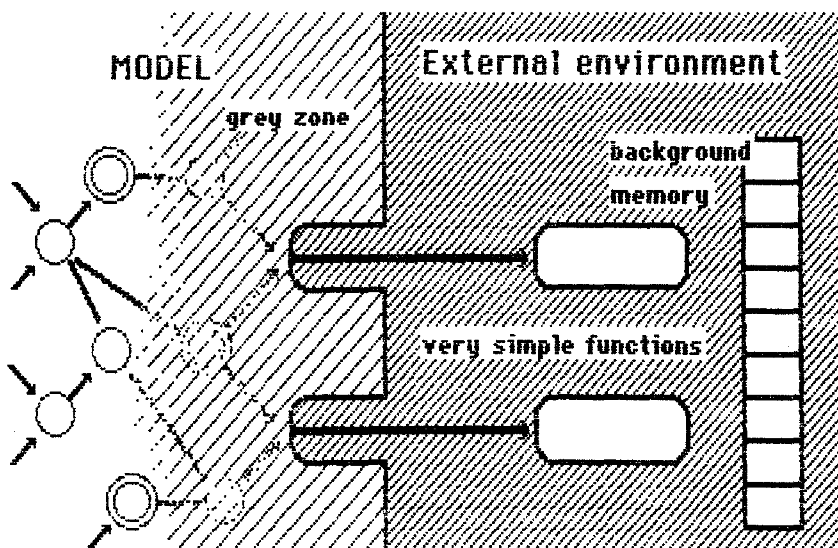


fig. 26 grey zone

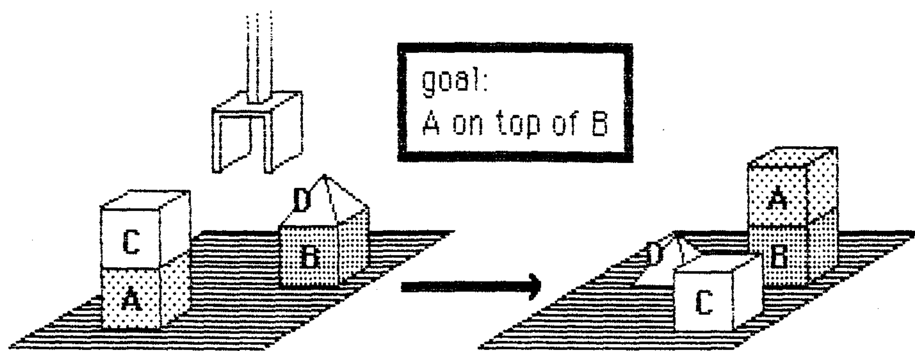
Chapter 10
Multiple use of knowledge

When designing a flowchart for a program, there usually are sequences of many similar operations to be carried out one after another. Even a computer programmer with little experience will not write them all out, but writes a loop, which tells the computer to perform a fixed set of operations and then loop back and perform them again, over and over, until some condition is satisfied ([4] p.149). And the body of the loop - the fixed set of instructions to be repeated - need not actually be completely fixed. It may vary in some predictable way.

Because of the structure of SAGE, it is worthwile to devote an entire chapter to the question how a designer should handle loops. I hope that this will be clear at the end of this chapter. However, it should be noted that in consultational programs the similarity in the instructions is usually not high enough to use loops effectively.

§1. In what kind of situations do we encounter sequences of similar operations?

I will discuss 4 kinds:
1) Search: until you find the thing you want, you carry out a series of operations. Those operations are usually the ones needed to determine whether you found what you want. The loop is aborted upon success or when there are no things left to examine. Usually, we are not interested in the results of the unsuccessful executions of the loop, we just want to find something.
2) Iterative use of knowledge, (very similar to 1)): when we have a data structure composed of several elements, which should all be processed in about the same way. The loop is aborted when there are no more elements to process. An example in KNOWBODE: we have a frequency-transfer function composed of several factors, which should all be examined for their break points, asymptotes etc.
Usually, we are interested in the results of every execution of the loop.
3) Recursive use of knowledge: loops may be nested; a special kind of nested loop occurs when an inner loop is quite similar to the outer loop. Thus, execution of the instructions in the outer loop will involve execution of a similar set of instructions which again can involve execution of such a set, etc. etc.
This is called recursion. Such program structures are typical - in fact they are deemed to be good programming style ([4] p.149, see also Hofstadter [5] (March 1983) p. 16). An example on the use of such recursive procedures is the execution of this goal tree (on the next page) (Winston [19] p.39):



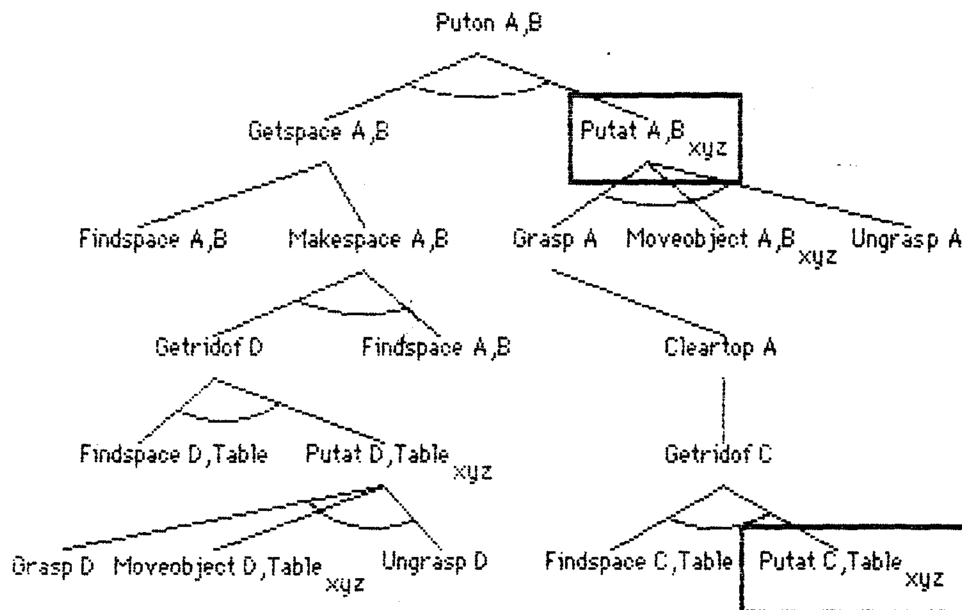


fig. 27 goal tree. Branches joined by arcs are under AND nodes, other branches under OR nodes.

In order to PUT block A AT the position on top of block B, the set of instructions needs to invoke a similar procedure, PUT block C AT the position on the table. (And if there were any blocks on top of block C, again similar procedures would have to be called).
Notice that 1. execution of the outer loop cannot be completed until the inner loops are finished, and 2. in order to work, the recursion has to bottom out somewhere.
4) Situations similar to 1) and 2): there is a sequence of operations which we want to use a number of times, but not in one loop. We want to use the sequence of operations, then do something else, and then do the specified sequence of operations again.
In computer programming we speak of a subroutine, a more general notion than loop ([4] p.150). Subroutines can call each other by name, and thereby express very concisely sequences of operations to be carried out. This is the essence of modularity in programming (compare this with the use of RULES in SAGE!).

§2. What possibilities are there in SAGE for loops and subroutines?

AREAs may be processed more than once in a program run, by using the the action effects RESTART or REASK. There are a few problems however. These operations only have an effect on the area within which they are

used: other AREAs which are CONSIDERed by ACTIONs in this area will not be reexamined. Which means that AREAS cannot be used in the same way most subroutines can.

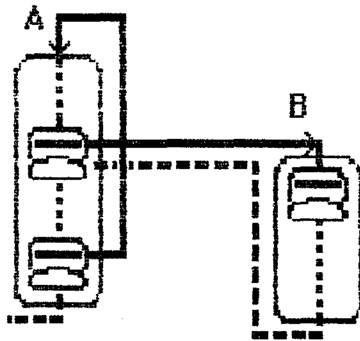


fig. 28 AREA CONSIDERed inside an AREA containing loop

See fig. 28 : if the procedure contained in AREA B is also to be used again in the second or later executions of AREA A, it cannot be a separate area but should be included in AREA A.
Also, nested loops of this kind:

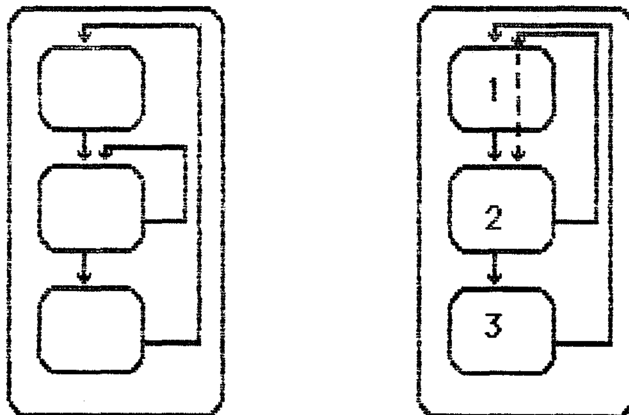


fig. 29 nested loops

cannot easily be contained in an AREA. Loops always start at the first ACTION.
One may however use preconditions on the ACTIONS in part 1, which will prevent their execution in case they are processed in the course of the inner loop (difficult! and not understandable to the user).

What about the elements of the RULE-based part?
ASSERTIONs and OBJECTs are investigated only once: when facts have been given a value, they keep that value ("truth maintenance", p. 10). But if the AREA to which they belong (i.e. in which they are declared in the source listing) is restarted, they are returned to an unevaluated state.
You might wonder how facts can get a different value upon reinvestigation, since the procedure to examine them is deterministic and not dependent on the fact that they have been returned to an unevaluated state. This is possible because the leaves of the goal tree, where the tree receives values, are made of QUESTIONs and FUNCTIONs. Questions may be answered

differently by the user and the calculation of functions <u>can</u> depend on what happened in the past.
RULEs are custom-made (by the programmer) for every OBJECT or ASSERTION*. Since facts retain their values, RULES to establish them need to be used only once in the investigation process.

So in SAGE our only** means of creating loops is to use RESTART or REASK in the program.
But all kinds of loops can be created in the external environment, using normal programming techniques. These loops unfortunately are not visible to the user. Unfortunately? Sometimes it is not necessary or desirable that the user should see everything, so we still have several possiblities.

§3. Then how should we handle situations with loops?

1) *Avoid the* <u>*need*</u> *to use loops if possible, in SAGE.*
We can avoid search situations if we have the knowledge where to look (this is obvious). An example: by putting all break points in a list in the external environment in size order, we <u>know</u> that the first break point which is larger than the current one is the next in the list. Sorting the break points (uninteresting to the user), has been done by a PROCEDURE (<sort>) in the external environment***.

Sometimes we can avoid the need to use recursion by rearranging the sequence of operations, but then there will still be (nested) loops. See an example .

2) *Don't use loops.*
(Why go through a lot of trouble, if not using them is easier?) When you are working in the RULE-based part of the model, you must realize that if you want the program to use part of the goal tree <u>again</u>, you're asking for trouble. See also chapter 8 where you were advised to avoid evaluation of the facts in a goal tree until you are sure that they have the desired value. Usually, the easiest thing to do is just to program another similar tree (with slightly different names, like <factorX> instead of <factor>).

For AREAs the situation is somewhat different. First, it is rather easy to use RESTART to introduce loops, and second, let us see what happens if you don't use them: suppose I had "folded out" AREA <examine_intervals into AREAs for every interval, I would have 10****AREAs of about the same size as <examine_intervals> plus the associated facts and rules! The model's source listing is a lot bigger, but <u>maybe</u> the program is faster

*** One might argue, as in the beginning of this chapter, that in <u>consultational</u> programs the rules are too specific to be generalized very much.**
**** The <u>user</u> has, in a dialogue with the program, explicit commands available to reinvestigate an AREA! This means that it is also possible to make a command file which contains these instructions. Such a file could then be started by the user through "DO filename", <u>upon a request by the program</u>! This is not an elegant solution.**
***** We could also have the break points in a list unsorted, and use a FUNCTION to search for the next break point and retrieve its value.**
****** because I allow the FTF to contain up to 10 factors, which gives at most 10 intervals between break points.**

(RESTART seems to slow him down). But what is really a problem is the fact that when the user asks for "LIST DONE GOALS" or "SUMMARISE" he will see 10 times the text associated with the area <examine_intervals> <u>plus all the actions with their associated text</u>. The user 1) will be flooded with text (which he can't interrupt! and 2) will probably not understand why the program doesn't simply say "I did this (10) times". I think that for AREAs the use of loops is not very bad.

3) *Use RESTART/REASK and the external environment*

I already pointed out some restrictions on the use of loops through RESTART/REASK (reference to AREAS outside the loop can be made only once, loops always begin at the first ACTION in the AREA). I also mentioned that the model does not remember that that it has executed a loop before (<u>after</u> the RESTART or REASK command). If a loop is restarted, it is the user who must notice that the same questions are asked again, and who may then give different answers (by using "REASK" the questions which are reasked are restricted to explicitly mentioned ones only). Or, the model should inform the external environment that it is going to restart an area, and there this information can be held in memory.

An easy way to do this, is to store the number of executed cycles of a certain loop in background memory. This is the way used in KNOWBODE: for instance, in AREA <xmine_low_freq\break> the number of factors examined (the number of executed cycles of the loop) is stored in position <factor_cycle> in memory. Such a position must be initialized properly: we already discussed that in Chapter 7.
It can be used by FUNCTIONs to figure out what factor to examine next:

```
RULE obtain_factor :
"takes a factor out of the FTF"
        factor IS next_factor( FTF,
                    factor_cycle)*
```

When the AREA is restarted, the number of cycles must be increased by one:

```
ACTION test_next_factor : "" **
        CALL increase( factor_cycle, 1.0)
        ALSO
        RESTART
        PROVIDED another_factor
```

The results of the execution of a loop which must be remembered, possibly intermediate results of calculations, must also be stored in the external environment. This is done in

```
ACTION add_break_point :
"finding the break point of a factor (if possible) "
"and adding it with the factor to a list"
        CALL add( break_point, cause_factor, list)
        PROVIDED break_point_exists
```

* there is no explicit need to show <factor_cycle> in the argument of the FUNCTION. This was done so the designer might easily see where the number of cycles is stored

and

```
ACTION memorize_partresult2 :
"putting some results in background memory in "
"order to be able to remember them in the next "
"cycle in this AREA"
        CALL put_in_memory([bufferY],
                      order_lowest_term_H,
                      coeff_1st_term_|H|,
                      coeff_2nd_term_|H|)
```

This way can be summarized as:

1) execute the sequence of operations
2) store (intermediate) results of calculations which are necessary for future cycles in background memory and/or produce output
3) erase everything in the AREA (RESTART)
4) and start again (using stored results).

It is a method which is very suitable for <u>iterative</u> use of procedures.

However, in the case of <u>recursion</u> as demonstrated by fig. 27 on p. 81 this method is not applicable, if we were to encode this goal tree into RULES for SAGE and make use of recursion for the procedure PUTAT. The "high-level" goal (PUTAT A,B_{xyz}) cannot be reached until the "lower-level" goal (PUTAT C,$Table_{xyz}$) has been attained. But this lower-level goal wants to use the same procedure (RULEs) as the high-level goal and has to wait until the AREA is cleared. This will not happen as long as <PUT_A_AT_B> has not been evaluated. Here we have a definite deadlock situation.

One might argue that reordering could solve the case:

1) first investigate and evaluate <PUT_C_AT_Table>
2) then clear the area, but retain the results (in the real world)
3) then investigate and evaluate <PUT_A_AT_B>.

This is a good solution, but I am not sure that it is always possible to order the steps of the solution in advance. What can you do in such a case? For the sake of completeness, I will discuss a solution in Appendix E.

4) *Use loops in the external environment*

We have already seen an example of loops in the external environment: sorting all break points in a list requires loops, which are easily encoded in FORTRAN or other computer languages.

I will discuss here another example, which is not implemented, but has interesting features.

If there was a FUNCTION <make_list>, which could act on <u>lists of factors</u> ("polynomials" like FTF) and make new lists of all factors which had a certain feature to be specified in the arguments, and which would return TRUE if there was at least <u>one</u> element in that list, then I could make ACTIONs like

```
ACTION xmine_trivial_factrs :
"does .... to trivial factors"
        (actionbody)
        PROVIDED I did make_list(of_factors_from FTF, with type_feature,
                    equals, order_0_or_time_lag, and_put_into list_2)
            AND I did make_list(of_factors_from list_2, with x_feature,
                    equals, 0.0, and_put_into list_3)
            AND I did make_list(of_factors_from list_3,
                    with variable_feature, equals, no_variable ,
                    and_put_into list_4)
            AND I did make_list(of_factors_from list_4, with mu_feature,
                    equals, 1.0, and_put_into list_5)
```

where "I", "did", "of_factors_from", "with" and "and_put_into" are NOISE words, and "type_feature", "x_feature", "variable_feature", "mu_feature", "equals", "order_0_or_time_lag" and "no_variable" are CONSTANTs to indicate to the function what should happen.
What is the advantage of doing it this way? Well, now the program can select all trivial factors in the course of one ACTION, use only one kind of FUNCTION to do it and will stop if there are no elements left to examine (see Chapter 8).
I don't even have to know how many trivial factors there are, it will work for any number (up to the length of the lists). Compare this with the use of algebraic expressions in the external environment, in Appendix D. There also, operations for more than a sinlge entity are done at the same time.

This whole chapter was about loops in a single program. We can also think of loops consisting of programs: a program is used on some data, and then it is used again on other data etc. Or a program is used on some data and then a slightly different (improved) program is used again on the same data, etc. etc.
Here the external environment can also be used to store "intermediate results". I shall not discuss this any further, but just notice that in this way the external environment may be used to incorporate "experience" into the program.

References

[1] d'Agapeyeff, A.
"Expert systems, fifth generation and UK suppliers", published by the National Computing Centre Limited, 1983.

[2] Cordier, Marie-Odile
"Les systèmes experts", (general article about expert systems) La Recherche vol. 15 no. 151, January 1984, p. 60-70. *(in French)*

[3] Elgerd, Olle I.
"Control Systems Theory", McGraw-Hill Kogakusha Ltd, Tokyo, 1967.

[4] Hofstadter, Douglas R.
"Gödel, Escher, Bach: an Eternal Golden Braid", New York: Basic Books 1979.

[5] Hofstadter, Douglas R.
"Metamagical themas" (about Lisp), Scientific American, February, March and April 1983.

[6] Kobashi, Y. and Kumar, A.
"Model mismatches in decision making by an expert system", Proceedings of the NGI-SION Symposion 2, 16/17 April 1984.

[7] McDermott, John
"R1: A Rule-Based Configurer of Computer Systems", Artificial Intelligence, **19** (1982) 39-88.

[8] Milne, A.A.
"The House at Pooh Corner", printed in Great Britain by Richard Clay (The Chaucer Press) Ltd, Bungay, Suffolk (1982).

[9] Nilsson, Nils J.
"Principles of Artificial Intelligence", Palo Alto, CA: Tioga Publishing Company, 1980.

[10] Newell, Allen
"The Knowledge Level", Artificial Intelligence **18** (1982) 87-127.

[11] Pavelle, Richard., Rothstein, Michael. and Fitch, John
"Computer Algebra", Scientific American, December 1981, 102-113.

[12] Rademaker, O.
"Dynamische Systemen", dictaat bij het college 3.720.0, dictaatnr. 3.316,
Technische Hogeschool Eindhoven (1978) (*in Dutch).*

[13] Reichman-Adar, Rachel
"Extended Person-Machine Interface", Artificial Intelligence, 22 (1984) 157-218.

[14] Ritchie, G.D. and Hanna, F.K.
"AM: A Case Study in AI Methodology", Artificial Intelligence 23 (1984) 249-268.

[15] Simon, Herbert A.
"The Sciences of the Artificial",MIT Press, Cambridge, MA, 1969.

[16] Simon, Herbert A.
"Search and Reasoning in Problem Solving", Artificial Intelligence 21 (1983) 7-29.

[17] Stefik, Mark et al.
"The Organization of Expert Systems, A Tutorial", Artificial Intelligence, 18 (1982) 135-173.

[18] Waltz, David L.
"Artificial Intelligence", Scientific American, Oct. 1982, p.101-122.

[19] Winston, Patrick Henry
"Artificial Intelligence", (2nd edition) Addison-Wesley Publishing Company, 1984.

[20] SAG01, *"An Introduction to the 'SAGE' Expert System"*, Knowledge Engineering Group, Systems Programming International (1982).

[21] SAG02, *"SAGE expert system Language Specification"*, Knowledge Engineering Group, Systems Programming International (1982).

[22] SAG03, *"SAGE User Manual (for SAGE version 1.3)"*, Knowledge Engineering Group, Systems Programming International (1982).

[24] SAG04, *"SAGE Installation Guide (for the PDP 11 range of machines operating under RSX-11M)"*, Knowledge Engineering Group, Systems Programming International (1982).