Designing debugging tools for Simplexys expert systems

Research Report
ISSN 0167-9708
Coden: TEUEDE

Eindhoven
University of Technology
Netherlands

Faculty of Electrical Engineering

Designing Debugging Tools for Simplexys Expert Systems

by
E.H.J. Philippens

EUT Report 90-E-234
ISBN 90-6144-234-5

January 1990

Eindhoven University of Technology Research Reports
EINDHOVEN UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
Eindhoven The Netherlands

ISSN 0167- 9708 Coden: TEUEDE

DESIGNING DEBUGGING TOOLS FOR SIMPLEXYS EXPERT SYSTEMS

by

E.H.J. Philippens

EUT Report 90-E-234
ISBN 90-6144-234-6

Eindhoven
January 1990

This report was submitted in partial fulfillment of the requirements for the degree of Master of Electrical Engineering at the Eindhoven University of Technology, The Netherlands.
The work was carried out from January 16, 1989 until December 1, 1989 under responsibility of Professor J.E.W. Beneken, Ph.D., at the Division of Medical Electrical Engineering, Eindhoven University of Technology, under supervision of J.A. Blom, M.E.E.

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Philippens, E.H.J.

Designing debugging tools for Simplexys expert systems / by E.H.J. Philippens. - Eindhoven: Eindhoven University of Technology, Faculty of Electrical Engineering. - Fig., tab. - (EUT report, ISSN 0167-9708, 90-E-234)
Met lit. opg., reg.
ISBN 90-6144-234-6
SISO 608.1 UDC 616-089.5 NUGI 742
Trefw.: anesthesie; patiëntbewaking / expertsystemen.

Summary

As a result of the increasing success in large scale integration techniques, rulebased expert systems have entered the application area of PC-systems. In connection with the research concerning Simplexys, a toolbox enabling the realization of real-time expert systems, some debugging tools have been developed. These debugging tools consist of a tracer, which makes it possible to 'trace' through the whole inferencing process, and an explain facility for examining the evaluation structure of the process. With the help of these tools, the users, usually knowledge engineers, can control and debug their rulebases or can prove the correctness and efficiency to experts in the domain problem-field.
The final implementation is working and has already been used to examine existing Simplexys expert systems.

Table of Contents

Chapter 1: Introduction

Because of fast developments in computer techniques, Artificial Intelligence (AI) came within the range of PC-systems. A special field within the AI-science is the design of real-time rulebased expert systems. Most existing expert systems can't manage real-time processes because it takes too much time for them to look up in their complex LISP-program-structures the knowledge they need.
At the division of Medical Electrical Engineering of the Eindhoven University of Technology a real-time Expert System Toolbox called Simplexys is under development for several years now [Blom,1988]. This group is performing a study in cooperation with the Department of Anesthesiology of the University of Florida in Gainesville.
Simplexys was mainly designed for monitoring tasks in medical applications, where efficiency and performance are of primary concern. Simplexys applications in development are an expert system for intelligent alarming for a anesthesia machine and a blood-pressure controller. The controller, which was developed by P. Hoogendoorn [1989] is now being tested at the Catharina Hospital in Eindhoven.
When testing these first real-time Simplexys expert systems, the need arose to examine the inferencing processes more closely. The knowledge engineer (or rulebase programmer) wanted to see how these expert systems really function in a real-time environment and whether he needs to add to or to correct the original rulebase. In other words, there was a need for some debugging tools to simulate the process and look inside the 'expert system black box'.
In this report we shall discuss what possible errors can occur in real-time expert systems and how and with what kind of debugging tools we can detect and locate them. We also shall describe these tools.

Chapter 2: **Expert Systems and debugging**

2.1 Expert systems: general approach

There are many definitions for the term '**Expert Systems**' [Harmon,1987; Rolston,1988; Aleksander,1986], but in my opinion the most general and complete one is:

'An expert system is a computer application that solves complicated problems that would otherwise require extensive human expertise. To do so, it simulates the human reasoning process by applying specific knowledge and inferences.' [Osterweil,1983].

To build up an expert system the expert system designer or knowledge engineer first needs to acquire the knowledge from one or more domain experts and he has to 'transport' that knowledge into a socalled knowledge base.
This process of acquiring the knowledge and building up the knowledge base (according to the system's knowledge representation) is called knowledge engineering. If the specific knowledge is represented as production rules in the knowledge base then we call the resulting Expert System a rulebased Expert System. The big advantage of expert systems above normal conventional programs is the flexibility of those expert systems. By flexibility we mean that when the surrounding of the expert system is changing we only have to add new or subtract old rules in the rulebase.
There exist two types of expert systems: Analysis Systems and Planning Systems. The goal of planning systems is a search for the best path between the current and a specific future state of the process. The goal of analysis systems like the Simplexys expert systems, is an analysis of one specific static situation only, although some earlier results may need to be remembered to achieve this analysis.
In this field of Artificial Intelligence some successful results have already been achieved, for example the medical expert system NEOMYCIN [Hasling,1984], which is a new version of the already existing MYCIN [Shortliffe,1976]. This expert system can be used by doctors who try to determine what kind of bacterial disease a patient suffers from. By asking questions to the user, this expert system tries to formulate a diagnosis about the disease and will propose a therapy as well. Some general ideas about expert systems and 'debugging' were taken from this expert system.

The big problem of expert systems is that we can distinguish several kinds of users who will use the Expert Systems in different ways:
-› The Testers: They attempt to verify the validity of the system's behavior.
-› The Tutors: They provide additional knowledge to the system or want to modify knowledge already present in the system.
-› The Pupils: They seek to rapidly develop personal expertise relative to the subject domain by extracting organized,

distilled knowledge from the system.
-› The customers: They apply the system's expertise to a specific real task.

Depending on the kind of user the Expert System Toolbox will be used differently and there will be a different need of certain aids/tools. On the contrary, for traditional software systems we only recognize one kind of user i.e. the customer.

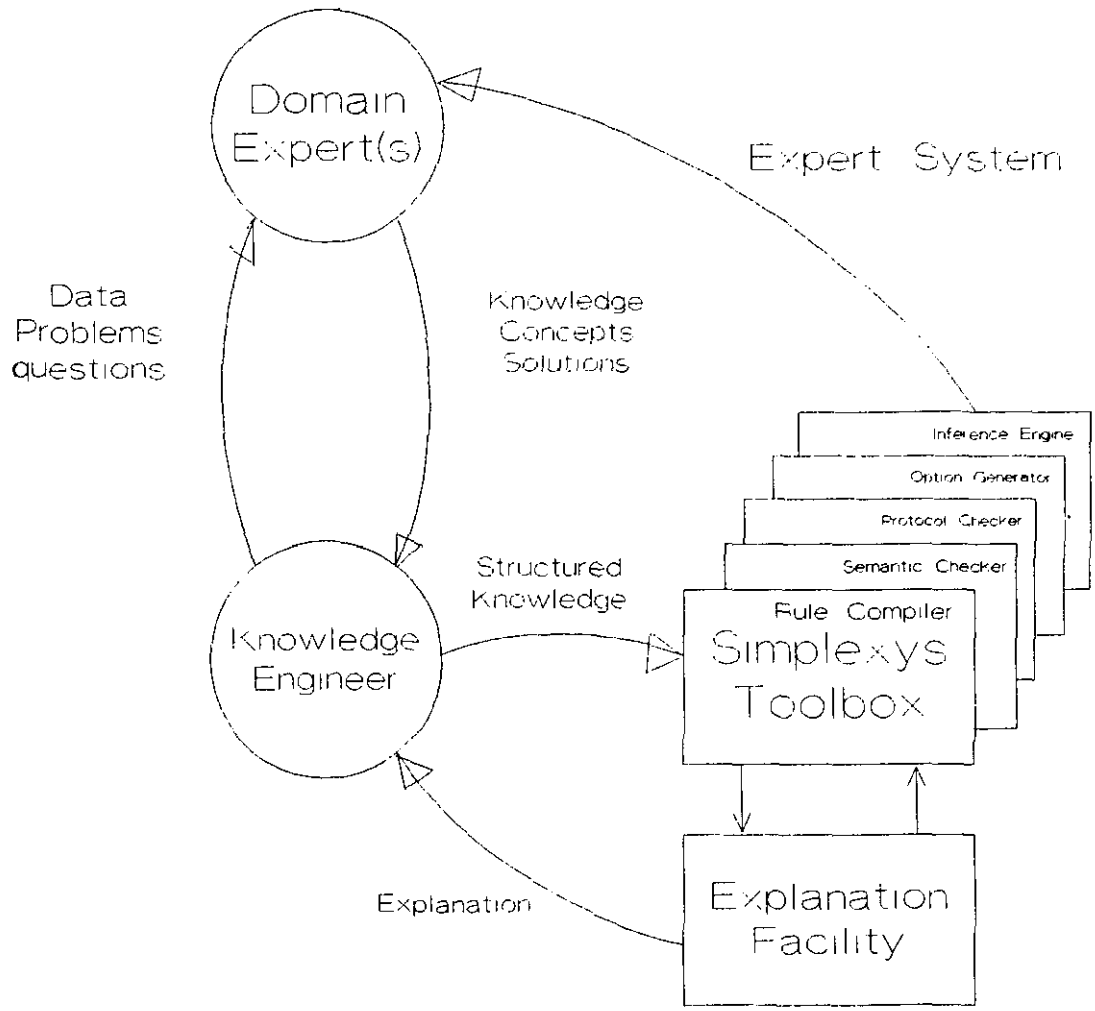


Figure 2.1: Simplexys Expert System Toolbox

Figure 2.1 shows how Expert Systems are made with the Expert System Toolbox Simplexys. In contrast with MYCIN, Simplexys assumes a fixed knowledge base which means that no rules can be added dynamically. So rules, links and data can be stored in known memory locations which results in no searching during the inferencing process at all. Another contrast with most expert system toolboxes is that Simplexys can be efficient in a static and in a dynamic environment.
The toolbox contains a Rule Compiler for translating the

rulebase into an internal representation that can easily be handled by the Inference Engine. This Inference Engine implements the necessary reasoning ability and delivers a computer program or expert system which is able to reason about its specific knowledge. The toolbox also contains some rulebase checkers and an Explanation Facility for explaining the rules of rulebases in a graphical way.

2.2 Debugging in general

Because there are no theories or methods for debugging expert systems, we will have to examine what exactly the differences are between 'normal' programs and expert systems. Afterwards we only have to adjust the debugging methods/tools of conventional programs for expert systems by taking these differences into account.
Debugging is the process of determining and eliminating different errors (bugs) in a certain program. Debugging is an important and often underestimated part of the process of programming.
Normally every programmer who is designing a program, tool or system, will recognize several stages in his so called software life cycle:

A=>= Preliminary design
  The programmer is trying to formulate a correct definition of the problem. He also has to study the system and software possibilities and requirements.
B=>= Detailed design
  The programmer is trying to invent and implement an algorithm which can solve the problem.
C=>= Compiling and Debugging
  During this developing stage the programmer will try to compile his program. During the compiling the programmer may also have to debug his program
D=>= Testing and Debugging
  When the program is compiled correctly, the programmer is still not sure that his program is working satisfactorily: he has to test it.
E=>= Maintenance
  Because the environment where the program is used can change and because some users are allowed to change the requirements, the programmer may have to update the program. To do it effectively the programmer should take care of program readability in earlier stages.

The only difference in the software life cycle of expert systems is that it contains another stage: knowledge acquisition.
When we look more carefully at the software life cycle, we notice that a programmer will spend most of his time on debugging (according to [Tassel,1974] ca 50%). During this time he will make heavy use of the machine he is working on. Whilst quality control was primarily restricted to one phase of the software life cycle (stage C), it is assumed today that quality assurance should be performed in parallel with the construction

and application of the software, i.e. that each intermediate and each final product of the production process should be examined immediately after its completion. The programming errors we will meet during each developing stage and which we also should detect during that same stage are:

A=›= Errors in the problem definition because of lack in the understanding between the programmer and the user(s) (lack of complete and adequate specification).
B=›= Errors in the design because of unsolved or overlooked questions of strategy and algorithm development:
  B1-› Incorrect algorithm
       The program is not doing what the user expects it would do
  B2-› Errors in analysis because of incorrect solving of the problem.
C=›= Errors in using the language because of inadequate knowledge of the language:
  C1-› Semantic errors
       failure to understand how a command works.
  C2-› Syntax errors
       failure to follow the rules of the programming language
       These errors are often detected and rejected by the compiler.
  C3-› Execution errors
       failure to predict the possible ranges in calculations or failure to anticipate the ranges of the data.
  C4-› Transcription errors or handwriting slips
       Some of these errors will be detected by the compiler but most of them will not.
D=›= Errors in selecting appropriate tests
E=›= Errors due to incorrect adaptation of the program for more users.

Normally when a programmer is speaking about errors in his program, he actually means the errors which will be detected during the Compiling and Debugging stage.

2.2.1 debugging tools

Not every good programmer is also a good debugger. For good debugging a programmer not only needs a certain amount of knowledge about the problem-field, but he should also have the ability:
-- to think logically
-- to be creative
-- to observe attentively (attention to details)

After each development stage a programmer should use different debugging techniques. For example when a programmer is tracing Pascal code, his debugger should use variable names instead of memory locations and Pascal statements instead of machine instructions.

At level C and D of the software life cycle, the best method to allocate an error in the program is to follow the "hypothesize

and test" algorithm:
1-› Describe the problem
2-› Guess where the error could be
3-› Guess what might be going wrong because of that error
4-› Test your guess
5-› Refine and repeat the process until you have found the error
6-› Determine the fix
7-› Weave it in

This debugging process will be much easier and faster if we use debugging tools. Debugging aids are stethoscopes necessary to isolate the cause and location of an error. We distinguish three different categories of debugging tools:

A Snapshot tools
They give a picture of how the program or its variables look at a certain point in time.
Only two kinds of snapshot tools exist; listings (before execution of the program) and specific/non specific dumps (during the execution of the program).

B Dynamic tools
They show the program or its variables in operation. We distinguish two different kinds of dynamic tools:
--› tracers:
They indicate what statements are performed and in what order. In debugging there are two dimensions to be traced: space and time. With space dimension is meant the storage space of the program in the computer and with time dimension is meant the computation cycles completed during execution of the program. Usually the time dimension is the most important and longest tracer. But debugging aids should always allow the programmer to trace both dimensions.
--› variable displays:
While the program is running these tools show the value of one or more variables each time they change.

C Interactive tools
These tools offer the user broad powers to stop the execution of the program in arbitrary places and under broadly specifiable conditions. During such suspensions of execution, these systems allow the user to examine such internal status information as the values of variables. These systems enable the user to study error phenomena in minute detail, and they support the process of unravelling the causal threads leading to the first manifestation of an error which might have actually occurred long before.
We distinguish three kinds of these debugging systems: Systems which offer the user to suspend execution (1) at any specified program location (the breakpoint capability), (2) when any specified program variable changes value (the watchpoint capability), or (3) after some fixed prespecified number of program statements have been executed (the program stepping capability) [Osterweil,1983].

Because debugging is the largest program developing cost it is better to prevent bugs as much as possible. The best way to do this is to follow certain common programming rules [Brown,1973] like:

-› avoid questionable coding
   Use the simplest statements and do not try new features until you are sure they will work.
-› avoid dependency on defaults
   Computer manufacturers can change the defaults.
-› never allow data dependency
   do not expect data to be in a special form but instead check the data at input time.
-› always complete your logic decisions
   for example when you expect only two different values do not check only for one and if false assume it is the other one.

2.2.2 debugging and testing

Another important stage in the development stage is the Testing stage. A lot of programmers always mix this stage up with the Debugging stage, but Debugging is the part of the software developing process which is performed while the product is unstable while Testing is the final exercise carried out on the stable software product before releasing it. Or in other words: testing determines that an error exists and debugging tries to localize the cause of the error. Thus, there is some overlapping of the two stages.

We can improve the software quality by:
1-- reducing the number of execution errors
2-- improving the performance of the software
3-- improving the portability of the software
4-- improving the adaptability of the software
5-- improving the correctness of the software

Beginning programmers often feel only the program needs to be debugged. That is, once the program works for one carefully selected group of data, they believe it will work for all other data. However we can distinguish five different levels of program correctness:
1 no language syntax errors
2 no runtime errors like arithmetic overflow or division by zero
3 correct results for a typical set of valid test data
4 correct results for a typical set of valid and invalid data
5 correct results for any possible data both valid and invalid

For most conventional programs it is possible to prove the correctness of the program for level 1 to 4 and it is impossible to prove the correctness of the program for level 5. Large or complex programs or tools will have already problems by proving the correctness for level 4.

The first level of errors that should be detected during compilation of the program and not during testing are:

C1-- Some semantic errors like:
     -Improper use of built-in functions
C2-- Almost all syntax errors that can occur like:
     -Incorrect statement syntax
      like incorrect variable names, incorrect use of
      operators, misspelled keywords, undeclared variables etc.
     -Incorrect termination of program sequence (missing ends)
     -Invalid expressions
     -Improperly mixed data
     -Incorrect nesting
     -Incorrect matching of arguments and parameters
C4-- A few handwriting slips like:
     -Uninitialized variables

The errors that a compiler will not detect constitute a correct use of the language which is not correct for the application. They can only be discovered by examining the output and following certain testing procedures. For conventional programs these errors are for example:
-› Logic errors
-› Misspelled variable names
-› Incorrectly Initialized variables
-› Forgetting to reset a variable
-› Incorrect termination of program sequences
-› Array subscripting out of bounds
-› File and data formats unmatched
-› Incorrect use of boolean expressions
-› Counters too small
-› Incorrect termination of loop
-› Attempting to process data after end of file
-› Using the wrong version of a program

2.3 Debugging Expert Systems

After having discussed the way how debuggers are used in 'normal' programs and/or tools, we are now ready to examine why we need a debugging tool for expert systems and what makes them different from the normal conventional programming languages. The kind of errors that can occur in the rulebase or during execution of the expert system are quite similar to the ones of normal programs. So first of all, the expert system toolbox should contain compilers which check the rulebase on syntactical errors, some semantical errors and some execution errors.
When the rulebase is compiled successfully it is still possible that the Expert System is not doing what it should do. We can distinguish several shortcomings, typical for expert systems only, at this stage (execution errors) [Hasling,1984]:

a) Ignorance : a piece of knowledge is missing
b) Stupidity : a piece of knowledge is incorrect
c) Incompetence : current set of conceptual primitives is incapable of expressing a needed piece of knowledge
d) Superfluity : a piece of knowledge is supplementary and

will never be used.

e) formalism bug: one of the control sections has a bug or the set of available representations is inadequate.

The process of solving these problems can be divided into two different phases: a phase of determining/uncovering the mistake in the knowledge-base called debugging phase and a phase of correcting the incorrect knowledge called the knowledge acquisition phase.
Testing and/or debugging are the hardest tasks in developing an expert system [Pau,1987; Hendler,1988]. In comparison with procedural languages, rulebased systems are extremely hard to trace. Nevertheless, it would be desirable to be able to "reverse" the actions as if under instant replay. Unfortunately the tools needed for this task are not available (yet).
On the other hand most expert system designers know already intuitively where the 'bug' is without detailed tracing. In this case the eyes of the knowledge engineer are the most important debugging tools. So if a programmer wants to be successful in debugging, he should better be a near-expert in the problem domain as well.
Another big difficulty in debugging is to discover that problems or shortcomings still exist in the rulebase.
Taking all these needs and problems in consideration it would be of great use to have the ability to look at the whole "inside world of action" of an expert system, and at the proper level.

Before we can discuss how we have to change Simplexys or add some new "debugging" tools to the toolbox, we first need to know what the rulebase and the already existing different tools of Simplexys look like. We also have to examine what kind of errors already will be detected by these tools.

Chapter 3: The Simplexys programming language

The structured rulebase built up by the knowledge engineer must have a special format before it can be presented to the Simplexys Toolbox. This rulebase contains seven sections which must start with the following keywords:

```
1 DECLS     : declarations          ┐
2 INITG     : global initializations │
3 INITR     : run initializations    ├── Optional sections
4 EXITR     : run exitcode           │
5 EXITG     : global exitcode        ┘
6 RULES     : the rules
7 PROCESS   : the protocol
```

Because the first five sections are Pascal sections most of the syntax errors and some of the semantic errors that can occur in these sections will be detected by a Pascal Compiler.

3.1 The Declaration section DECLS

This section is optional and contains the Pascal declarations of all the variables, procedures and functions that will be used by initializations, exitcodes, TEST rules and THELSE DO's. The Inference Engine will include this section without any changes.

3.2 The Initialization sections INITG/INITR

These sections are optional and contain only Pascal code. The difference between the two sections is that the statements of the INITG section will be executed immediately after the system startup and the statements of the INITR section will be executed immediately at the start of each new run.

3.3 The Exitcode sections EXITG/EXITR

These sections are optional too and contain only Pascal code as well. The EXITG section will only be executed at the end of the last run and the EXITR section will be executed immediately after each run.

3.4 The declaration section of the rules

Section 6 must contain all the descriptions and definitions of all the rules. Each rule consists of two to four parts:

```
1-- rule header
2-- rule type
3-- initial value
4-- thelses
```

ad 1) rule header:
Contains the name of the rule and an explanatory text-string. For example:

Diabet_patient: 'Patient is suffering from diabetes'

ad 2) rule type:

Depending on the type of rule, evaluation of a rule can be achieved in three ways:

a) for the evaluation of the rule information is needed from outside the rulebase: primitive rules.

There are four different primitive rules:

FACT rules are assigned a certain value only at the beginning of the process. So they will never change value during the process. They can only be the value True,False or Possible. An example of such a FACT rule is:

```
Diabet_patient: 'Patient is suffering from diabetes'
FACT
```

ASK rules ask a question (=textstring) to the user and the answer will be the value of these rules (y(es)=True, n(o)=False and ?=Possible). An example of this kind of rule is:

```
must_Finish: 'You want to quit the process'
ASK
```

During the process we will see the message: 'Is true: You want to quit the process?'

TEST/BTEST rules contain one or more valid Pascal statements which will assign a value (True,False or Possible) to these rules (default=False). An simple example of a TEST rule is:

```
temp_too_high: 'the temperature is too high'
BTEST temp>40
```

MEMO rules 'remember' their value and therefore can only be changed by a THELSES (part four of the rule) in a previous run. An example of a MEMO rule is:

```
SETP: 'A setpoint has been defined'
MEMO
```

b) For the evaluation of the rule, values of other rules are needed. We call these rules EVALUATION rules. They contain an expression which is a combination of other rules (primitive or evaluation rules) and a number of operations. We have two different groups of operators:

Operators with one argument:

NOT v ::= if v = TR then FA else if v = FA then TR else v
MUST v ::= if v = PO then FA else v
POSS v ::= if v = PO then TR else FA

history operators: these operators are used as follows

```
RULE historyop (<numeric expression>)
historyop are '=','<>','>','>=','<' and '<='
```

The resulting value is first set to the result of RULE. Then, if the result is TR, the RULE's history counter value is compared, using the history operator, to the value of the numeric expression. If the comparison yields true, the result will be TR else the result will be FA.

Operators with two arguments:

There are five different operators which need two arguments: AND, UCAND, OR, UCOR and ALT (see figure 3.1).

AND/ucAND

w \ u	TR	FA	PO
TR	TR	FA	PO
FA	FA	FA	FA
PO	PO	FA	PO

AND: if u=FA then w is not evaluated

OR/ucOR

w \ u	TR	FA	PO
TR	TR	TR	TR
FA	TR	FA	PO
PO	TR	PO	PO

OR: if u=TR then w is not evaluated

ALT

w \ u	TR	FA	PO
TR	TR	FA*	TR
FA	TR*	FA	PO
PO	TR	PO	PO

* contradiction
ALT=logically equivalent alternative

figure 3.1: the operands AND,UCand,OR,UCor and ALT

An example of an evaluation rule is:

```
ShortDown: 'Too quick from fase 1 to 2'
Autofase1 < (45) and trig or not dumm
```

history expression

c) Another totally different rule-type needed to describe the dynamics of an expert system is the <u>STATE rule</u>. The values of these rules can only be changed by a STATE transition and their value can only be False or True. We will discuss these rules in section 3.5.

ad 3) <u>initial value</u> (not mandatory):
With this section we can give the rule a value (true,false or possible) for the first run

ad 4) <u>thelses</u> (not mandatory):
This section allows multiple consequences from a single rule evaluation. There are three types of THELSES: THENs,ELSEs and IFPOs. THENs are used to allow consequences if the result of the rule evaluation is TRue. ELSEs are used to allow consequences if the result is FAlse and IFPOs are used to allow consequences if the result of the evaluation was POssible.

A thelse must be followed by one of the next three possibilities:

a) a value (THELSE TR/FA/PO)
These THELSES allow multiple conclusions from just one evaluation. Under condition that the evaluated rule has a certain value, other rules will be assigned to certain values too. For example: "THEN TR: BIRD".

b) a goal (THELSE GOAL)
The argument, which must be a rule, is evaluated immediately. For example: "ELSE GOAL: BIRD".

c) a Pascal section (THELSE DO)
This kind of THELSES provides a "hook" to Pascal to manipulate data, to print etc. The Pascal statements are executed if the rule was evaluated to TR (THEN DO), FA (ELSE DO) or PO (IFPO DO).

An example of a rule containing four parts is:
Must_finish: 'You want to quit the process'
ASK
INITIALLY FA
THEN FA: Running

3.5 The process description section

The section 7 is mandatory and describes the dynamics of the process. The total analysis process is divided in a number of runs, during which the rule values do not change (only from undefined to a certain value). Each run can be divided in several subruns, which are evaluations of one GOAL rule. So we can distinguish a static environment during which the rules are evaluated at most once and a dynamic environment during which the rule values can change.
Because the Inference Engine is designed to work within a static environment, the dynamic environment needs to be translated into a sequence of static environments (see figure 3.2).

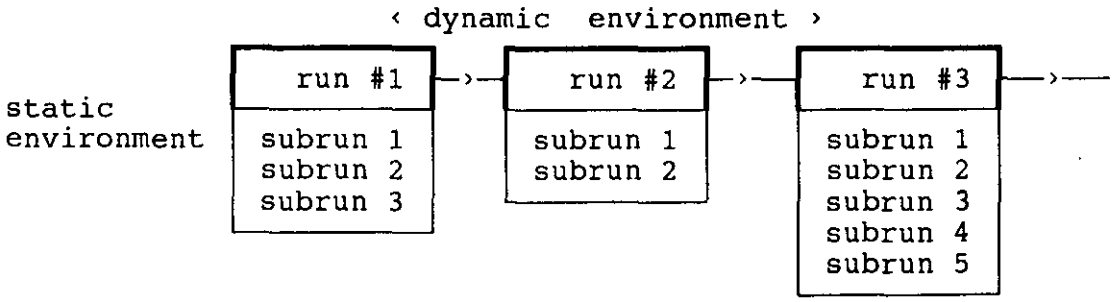


figure 3.2: An example of a process containing several (sub)runs

What Goals will be evaluated during one run mostly depends on what States are active (true) in that run. In the first run at least one State rule is initially true. The Process section contains all the state transitions, which have the following format:

ON **Trigger** FROM **Fromlist** TO **Tolist**

A State transition causes a change in active State(s); some States can become active (true) while others can become inactive (false).The State transition takes place if all the States in the Fromlist are active (true) and if the trigger evaluates to TRue. The process will end if no states are active anymore.
One way to analyze the dynamical behaviour of Simplexys expert systems, is by describing the process with 'Petri Nets' [Reisig,1985].
Figure 3.3 shows us a simple example of how to build up a process section by drawing a graph of the process. In this graph every circle represents a State rule and every connection between two circles represents a state transition (the example

is a simplification of the total debugging process of section 2.1).

```
States:
S1:'Describe the problem'
S2:'Guess where and what
the error in the program
is'
S3:'Determine how to fix
the error and weave it in'
S4:'Testing the program'
Triggers (for example ASK
rules):
Tr1:'New guess'
Tr2:'Satisfying guess'
Tr3:'Wrong consideration'
Tr4:'Error not solved'
Tr5:'Error solved'
Tr6:'Still errors in the
program'
Tr7:'Program is correct'

PROCESS
ON TR1 FROM S1 TO S2
ON Tr2 FROM S2 TO S3
ON Tr5 FROM S3 TO S4
ON Tr7 FROM S4 TO *
ON Tr6 FROM S4 TO S1
ON Tr4 FROM S3 TO S2
ON Tr3 FROM S2 TO S1
```

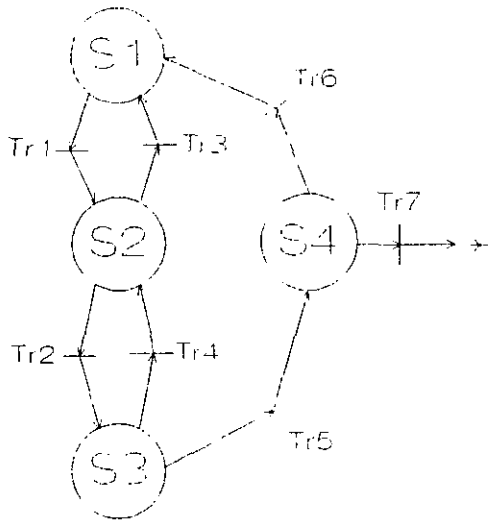


figure 3.3: An example of how to 'understand' triggers and states

Chapter 4: The Simplexys Toolbox

If we want to design debugging tools for Simplexys, we have to know exactly how Simplexys builds up a certain expert system. The Simplexys Toolbox already contains several Pascal programs or tools, which tend to incorporate restrictions that make them easy to use for certain purposes. Figure 4.1 shows us how to use the tools to build up a Simplexys expert system. Some tools already detect certain errors which we do not have to detect with the debugger anymore:

a) a Rule Compiler (Ruc.pas)
b) a Semantic Checker (Chk.pas)
c) a Protocol Checker (Pet.pas)
d) an Option Generator (Opt.pas)
e) an Inference Engine (Sim.pas)
f) an explanation facility called FACE (Face.pas + FaceExpl.pas)

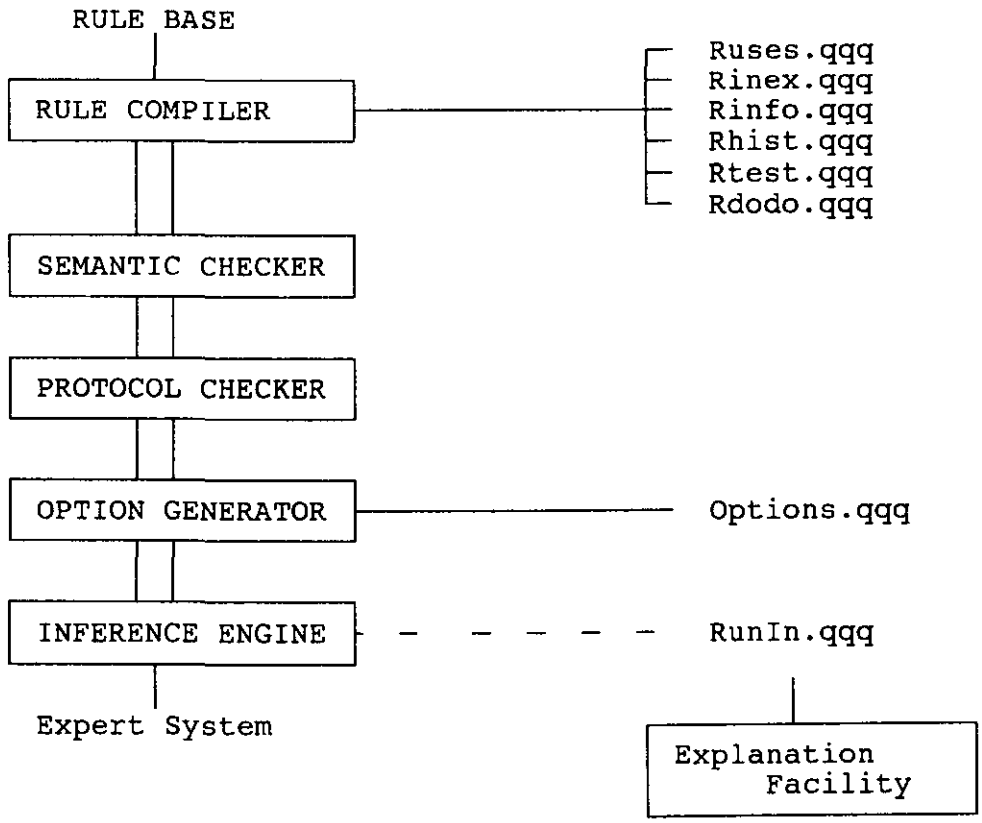


figure 4.1: How to build up a Simplexys expert system

4.1 The Rule Compiler

The Simplexys Rule Compiler translates the rulebase into an internal representation of six qqq-files:

1) Rinfo.qqq
   This file contains almost all the arrays and tables representing the rules and their mutual relationships.
2) Rtest.qqq
   All the test sections of the several defined test rules are translated into one function called "_FTEST".
3) Rdodo.qqq
   The DO commands in all the Thelses-Do statements are converted into one procedure called "_FDOS".
4) Rhist.qqq
   All the history sections of the used evaluation rules are translated into one function called "_FHIS".
5) Rinex.qqq
   This file contains the in the rulebase programmed initialization sections; INITR,INITG,EXITG,EXITR,EXITG. These sections are represented in this file as procedures with the same name.
6) Ruses.qqq
   Ruses contains the needed Turbo Pascal 'uses' libraries (specifically needed for programs written in Turbo Pascal).

The Rule Compiler also checks the rulebase for some semantic and syntax errors and gives an appropriate error message to the programmer.
The six semantic errors which can be detected in this stage are:
1) STATE rules are not allowed to have the value possible
2) At least one STATE rule must be initially true
3) FACT,MEMO,STATE rules cannot be used as goal rules
4) It is not allowed to THELSE FACT or STATE rules
6) In FROM,TO list it is only allowed to specify STATE rules

The compiler not only checks for syntax errors like mistakes in expressions, transition description lines,thelses etc, but also for errors like using more than one rule with the same name or internal (overflow) errors like using to many history checks.

4.2 The Semantic Checker

The Simplexys Semantic Checker performs several semantic checks on the file Rinfo.qqq and generates error messages if any errors are detected.
The six semantic errors checked by this program are:

1) Self-referencing evaluation loops:

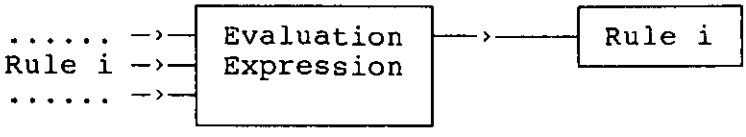


figure 4.2: A self-referencing evaluation loop

We get a conflict if we want to use a rule in its own

evaluation expression.

2) Thelses loops to itself:

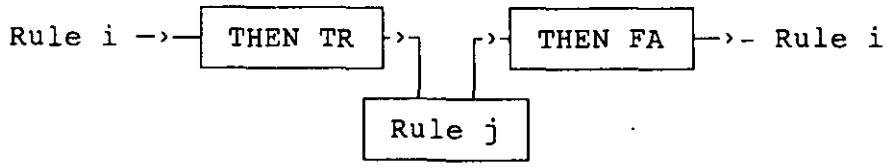


figure 4.3: A rule with a THELSES loop to itself

Conflict: If rule i evaluates to true then rule j becomes true and wants to set rule i to false.

3) Conflicting Thelses:

Two possibilities:

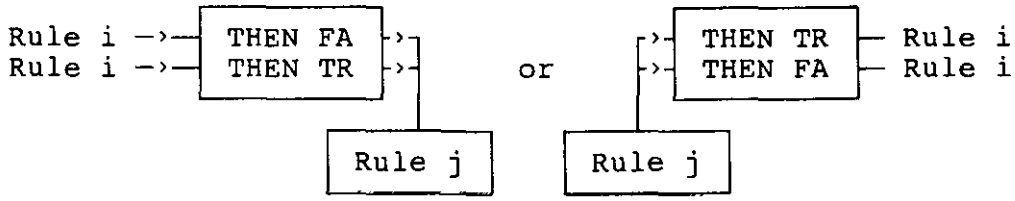


figure 4.4: An example of a conflicting THELSES

We get a conflict if we try to set a rule to true and to false at the same time.

4) Thelses to successors (semantic conflict):

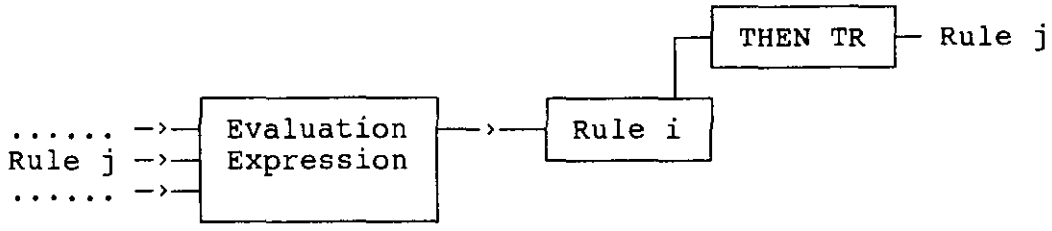


figure 4.5: A rule with a THELSES to a successor

Rule i needs for its evaluation rule j; it will try to evaluate rule j. After this evaluation rule i will set rule j to true even if the result of the evaluation of rule j was false.

Note: There will be no conflict if the rule is a MEMO or STATE rule.

5) Thelses to predecessors (semantic conflict)

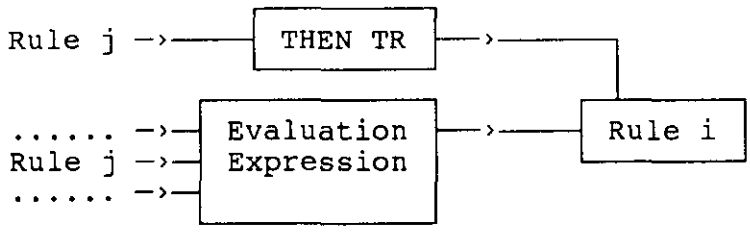


figure 4.6: A rule with a THELSES from a predecessor

Conflict: If we try to evaluate rule i, rule j needs to be evaluated as well. If a certain rule is evaluated, all its THELSES will be executed immediately. This means in this case that whatever the result of the evaluation of rule i was, if rule j became true then rule i will be set true.

Note: There will be no conflict if the rule is a MEMO or STATE rule.

6) Unconnected non STATE rules
A warning will be generated if a certain rule will never be used in the process:
---the rule is not used in any expression
---the rule is not THELSEd by any rule
---the rule is not a trigger rule

4.3 The Protocol Checker

This checker tries to find the errors in the process description part or protocol of the rulebase (rinfo.qqq). It checks the protocol on three different types of errors:

A--Syntax errors:
1. 'No start states'; no rules are initially true
2. 'No end states'; no ON statement has an empty TO list
3. 'Conflicts at states'
   Two transitions have equal From lists and the same trigger.
4. 'Empty prestate'
   Each State must be in at least one To list.
5. 'Empty poststate'
   Each State must be in at least one From list.

B--Topology errors:
Topological checking is performed in order to find errors in the way States and transitions are connected.
6. 'Self loops'
   The From list and the To list of a On statement are not disjunct.
7. 'Identical ON statements'
   Two On statements have the same From list and the same To list.
8. 'Identical STATEs'
9. 'Net part not connected to start state'
   To each State there must be a forward path from one of

the start States (result of the error: superfluity).

10.'Net part not connected to end-state'
From each State there must exist a path to an end-State.

C--Dynamic errors:
If we analyze the dynamic behavior of the protocol by constructing all reachable contexts, we can discover several dynamic errors.

11.'Deadlock'
There is a firing sequence resulting in a context where no further change of State is possible.

12.'Non safe state'
A non safe state is a State that becomes true due to firing of a transition while that State was already true before firing.

13.'System cannot stop'
There is no firing sequence so that only end-States are true.

14.'Not all transitions can fire at least once'

15.'Conflicts'
Conflict when the From lists have a non empty intersection.

4.4 The Option Generator

With the Simplexys Option Generator we can determine several run-time options for the Inference Engine (for example the choice between real-time and simulated time).

4.5 The Inference Engine

The Simplexys Inference Engine actually builds the expert system by compiling the several qqq-files and inference processes into one 'program'. The expert system is now ready to run.

We can divide one execution of a Simplexys Expert System into three main parts and several small steps:

A One-time initialization:
1--Initialize the time
2--Execute INITG
3--Initialize all the rules with an 'INITIALLY'-section
4--Determine all the FACT rule values
5--Execute the THELSES of all ASK, TEST and EVAL rules with value<>UnDefined

B Execute runs
1--Execute the 'THELSE'-section of all FACT and MEMO rules
2--Execute INITR
3--Execute the THELSES of the active STATE rules
4--Evaluate all the state transitions
5--Update the history of the MEMO and STATE rules
6--Execute EXITR
7--Undefine all ASK, TEST and EVAL rules
8--Goto B if any STATE rule is still active

C Finish
9--Execute EXITG

4.6 The explanation facility FACE

An explanation system must be capable of explaining, to some significant degree, the expert system's operations to anyone who understands the framework for explanation and who understands the domain subject. There are three types of explanation:

1-› Retrospective reasoning
Trying to explain what the system already has done.
2-› Counterfactual reasoning
Trying to explain what the system prevented from using rules that would have established specified facts.
3-› Hypothetical reasoning
Trying to answer the question: "What would happen if ...."

Nowadays explanation systems mostly use the explanation of type one. They are often designed in such a way that they can answer three types of questions:

-- WHY is it important to determine that.....
Asking WHY questions is the same as moving in the direction of the root or goal in the evaluation tree.
-- HOW was it established that.....
Asking HOW questions is the same as moving to the leaves or primitive rules of the evaluation tree.
-- WHEN was it established that.....
In contrast with the other two types of questions for WHEN-questions we are not moving in the evaluation tree, but just asking for the time or the history of that rule.

Mostly explanation system are used for assisting in debugging, for testing the expert system, for learning and teaching how to use/make an expert system and for updating the rulebase.

For Simplexys there already existed an explanation system called FACE which was developed by de Hair [lit. 2]. This 'FACility for Explaining simplexys expert systems' tries to answer the questions by just displaying the graphical presentation of the rule concerned with that question. It uses colors to express the run-values of all the rules used in that evaluation tree. There is also the possibility to 'trace' (run by run) through the whole process.
But several considerations lead to the decision not to use the explanation facility FACE as a part of our debugging tool:
1)
FACE does not function for more complex rulebases. If the rulebase contains Thelse-Do statements or initialization procedures FACE will crash.
2)
FACE uses colors to display the values of the rules in the rule evaluation tree. This means that a color monitor must always be available.
3)
FACE uses several smart algorithms to sort the rules of an

evaluation tree in such a way that it is easy to draw the tree. For debugging, however, we want to display the rulebase as the knowledge engineer programmed it.
4)
FACE just displays the result of one run. It is possible to trace through the whole process but you cannot recognize certain patterns and you can easily get 'lost' in the process. Especially when you have a rulebase with a lot of states, it takes a lot of time to find out what states are active and what rules were evaluated during that run.
5)
FACE uses the rule-names for representing the rules in their evaluation tree but it is possible that the knowledge engineer chooses very long names for his rules. This will make the display of the tree quite broad and it is now possible that the lines are longer than the predefined window.
6)
FACE also contains a simplified copy of the Inference Engine for simulating and tracing the process. But this means that if the original Inference Engine program is updated, we also need to alter FACE in the same way (which is the reason why FACE was not working correctly anymore).

Because we did not want to use FACE as our explanation system we had to look for other existing explanation system. In the literature we found several built-in explanation systems for all kind of expert systems which all try to answer the earlier mentioned questions [Hasling, 1984; Neches, 1984; Khoroshevsky,1985]. However it turned out that the implementation and layout of these systems depend on the kind of expert system to be explained and the degree of knowledge of the user.

4.7 What the debugger should do

As mentioned in section 2.3, it is possible that there are some errors in the rulebase which cannot be detected by any of the already discussed Simplexys checkers. Most of these errors will result in one of the shortcomings mentioned in the same section:

-› Incorrect expressions in the evaluation rules
   This handwriting slip can result in a Logic error, for example:
   You typed in for the evaluation part of a certain rule
   ATRI AND EQV3 ALT AL60
   but you actually meant
   ATRI AND (EQV3 ALT AL60)
   result of this mistake in the rulebase: Stupidity
-› Handwriting slips in the names of rules can cause new unwanted rules in the rulebase (misspelled variable-names).
   result of this mistake: Superfluity
-› Incorrectly initialized rules or just forgotten to

initialize a rule
-› wrong or no INITIALLY-section
-› execution errors
-› using the wrong version of the expert system

For these errors or to detect shortcomings of the rulebase, we need to design debugging tools for Simplexys. As discussed before one of these debugging tools has to be an explanation system.

Chapter 5: Design of the Simplexys debugger

As we already saw, a lot of errors will be detected by the different Simplexys checkers. The best way to debug errors that took place in real-time processes (=execution errors) and the best way to check the rulebase on some shortcomings (see section 2.3), is to simulate and trace the whole process again. Simulation gives us also the opportunity to examine the rulebase more carefully and to show internal variables (possibility to look inside the black box). The problem here is to design a good interface for showing the whole inside world of action to some understandable degree for the knowledge engineer.
In many cases the process consist of many runs (if one run takes 5 seconds, and an operation lasts 6 hours, the number of runs is 4320) and is it impossible to simulate and debug the whole process at once. So we split up the debugging process into a chain of several simulations where each simulation block can be examined by the debugging tools (breakpoint debugger,see figure 5.1).

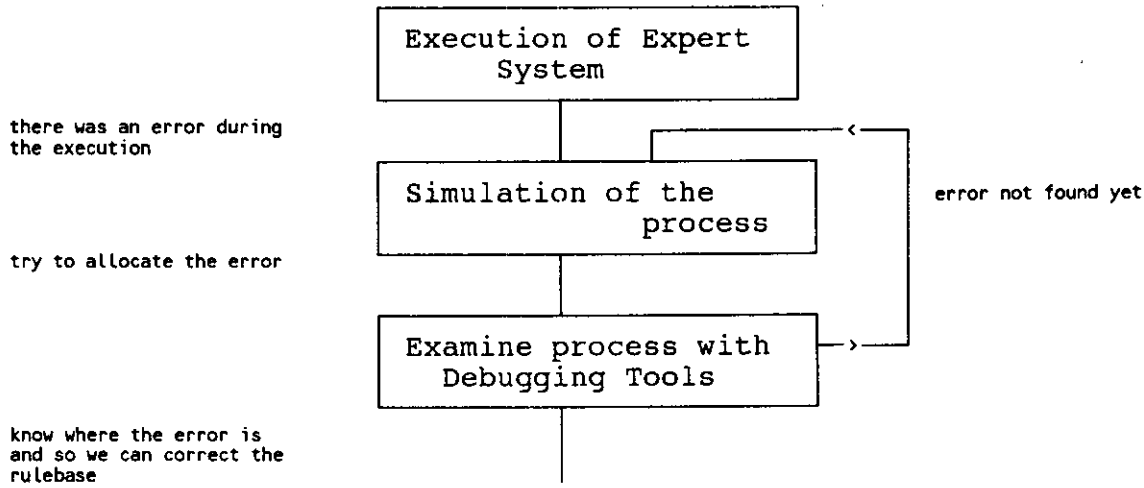


figure 5.1: The total overview of the debugging process

So we can distinguish two different parts in the whole debugging process: simulating and examining the simulation with debugging tools.

5.1 Simulating the inferencing process

Before we can simulate the real-time process, we have to solve a number of problems. First of all, the data used during the process is not available anymore at the moment of simulating. In Simplexys there exist several ways to acquire data from outside:

A) The sections INITR,INITG,EXITR,EXITG of the rulebase are Pascal sections and can therefore contain read(ln) or other input/output (IO) statements.

B) The RULES section:
-1- FACT rules acquire their values before the inferencing process starts (they obtain their value by asking a question).
-2- ASK rules also need real-time answers.
-3- TEST rules need externally supplied data.
-4- EVAL rules combine other rules in their expression
It is possible that they use history expressions in their evaluation expression. These history expressions can contain Pascal variables (for example: "BOIL › (12*temp)" with temp = variable of type integer).
-5- The 'THELSE DO' section of the rules provide a 'hook' to Pascal and therefore can contain IO-statements.

A solution to all these problems is to change the Inference Engine in such a way that it dumps the values of all the assigned FACT rules (two-bit value) in the beginning of the real-time process to a certain dumpfile. We also need to dump, for each run, the starttime (32 bit value) of that run and the results of the in that run used TEST, ASK rules and history expressions (two bit value). So the dumpfile will have a length of approximate

(number of runs) *
{((number of evaluated B1,B2,B3,B4,B5) + 7) div 8 + 4} bytes

The advantage of this solution is that the simulator will now look almost the same as the Inference Engine with the following exceptions (see also enclosure 1 and 2):

A -› the Pascal sections INITR,INITG,EXITR and EXITG are not executed.
B -› the values of the ASK, TEST rules and history expressions are read in from the file made by the Inference Engine during the expert system's actual operation.
B5-› the THELSES-DO parts of all the rules are not 'evaluated'.
C -› for every new run the run-time is read in. During the simulation it is possible that a run lasts shorter than it really did during the real-time process. If we would not dump the start-time of a run, the history of the rules would be incorrect during the simulation.
* -› in order to save some memory, we will make the file ruses.qqq empty. For the simulator we do not need any special libraries (no graphics).

So we can distinguish three different Inference Engines:
1-› original Inference Engine or inference engine which generates expert systems that do not generate a dumpfile. It is not possible to simulate the processes of these expert systems.
2-› Inference Engine which creates a dumpfile. It is possible to simulate the processes of expert systems generated by this inference engine.
3-› Inference Engine as simulator/debugger. The Inference Engine is used to simulate and debug a certain process.

To get fast tracers, it is better to look at all the rule-values (except the FACT rule-values) of several runs around the problem-area and put them in memory, than simulating and tracing for every run alone. How big a simulation block may be, depends on how big the rulebase is and how much memory is available. The worst case is that the rulebase is so big that each block is just one run.
If the programmer knows almost exactly when the mistake took place, he can choose the simulation block very small and simulate the process until that specific time. If he knows nothing about the error or if he just wants to examine the process, he will choose the simulation blocks very large in order to have the ability to look at many runs at the same time.

5.2 Simplexys debugging tools

Now that we know how we can simulate the process, we can try to find the error or just examine the different simulation blocks more carefully. The best method to find an error in the rulebase is (see also figure 5.2):
A)
First of all try to find out when the error actually took place in the real-time process. Processes which lasted very long will be simulated in blocks. So before we can determine in what run the error actually took place we first have to determine which simulation block. The tracer should therefore not only allow tracing through the whole process run by run but also block by block. Because a Simplexys expert system is working time-dependently and because memory is limited, we will not allow the programmer to trace one or more blocks back in time. If we know when the error took place we can determine with the tracer what states were active at that moment.
B)
Secondly we have to examine what caused the error and how it took place. For these questions we need an explanation facility to look at the structure of the rules in question. The structure of the STATE rules will tell us what rules were evaluated first.
If we keep on going down in the total evaluation tree of one run, we will finally reach the primitive rules.
We will call this facility the tree option of the debugger. It is possible that after you examined the structures of some rules, you discover that the error took place sooner but still in the same block and so we need to go back to the tracer (back to A).
C)
At last we know now what caused the error and so we can update or correct the rulebase.

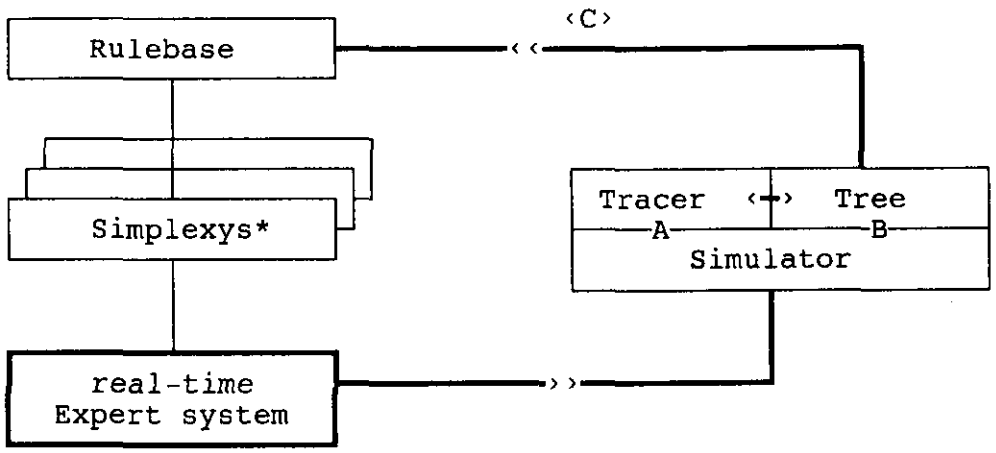


figure 5.2: The process of debugging the rulebase

So we need a tracer and an explanation facility for our Simplexys 'debugger':

5.2.1 The Tracer option

If we examine the rule-types more carefully then we can distinguish three groups of rules which only differ in the **number of different values** they can be during the process:
A-› FACT rules: just one value (TR,FA or PO)
B-› STATE rules: two different values (TR,FA)
C-› ASK, EVAL, MEMO and TEST rules: four possible values (TR,FA,PO,UD)
Of course each group needs a different approach for displaying their values:

Ad A) Displaying FACT rules

The FACT rules are evaluated only once in the process and they can have only the value True, False or POssible. So the value of a FACT rule can be displayed by just printing one string (' true',' false' or 'possible') in a rule-information-window (snapshot debugger).

Ad B) Displaying STATE rules

The STATE rules can only be FAlse or TRue and so we need only one character-line to display the values of the STATE rules; TRue = level 0 and FAlse = no level.

Figure 5.3 shows us an example of how to display a State rule.

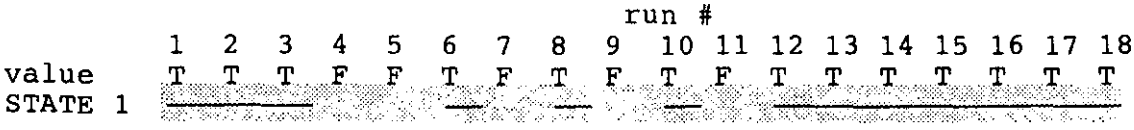


figure 5.3: An example of how to display a STATE rule

It should also be possible to display ASK, EVAL, MEMO and TEST rules in this way (TRue=level 0, FAlse/POssible/UnDefined = no level), because with this method we can display more rules. Because STATE rules determine what other rules will be evaluated during the run, it would be of great use to display all the rule-numbers of the active STATES at each run.

ad C) Displaying ASK, EVAL, MEMO and TEST rules

In each run these rules can have one of the four values: TRue, FAlse, POssible or UnDefined. There are several possibilities to display the values of these rules during the process:
A- with one character; for example T F P AND U
B- with colors; for example green red magenta and black (FACE)
C- with 'graphics'
Because we want to recognize certain patterns in the process and because we do not want to be dependent on the kind of monitor, we choose method C for displaying the values of these rules.

One way to display the values of these rules is using just three different levels:
False=level 0,Possible=level 1,True=level 2,Undefined=no level.
But now there are still several methods left to draw such a graph:

method 1:Draw one horizontal line for each level
method 2:Draw an almost 'closed' graph by drawing vertical lines when the level changes.
method 3:Same as method 2 only now when the value is undefined this method will remember the last level and will act like the UD-values do not exist.

An example of each method is given in figure 5.4.

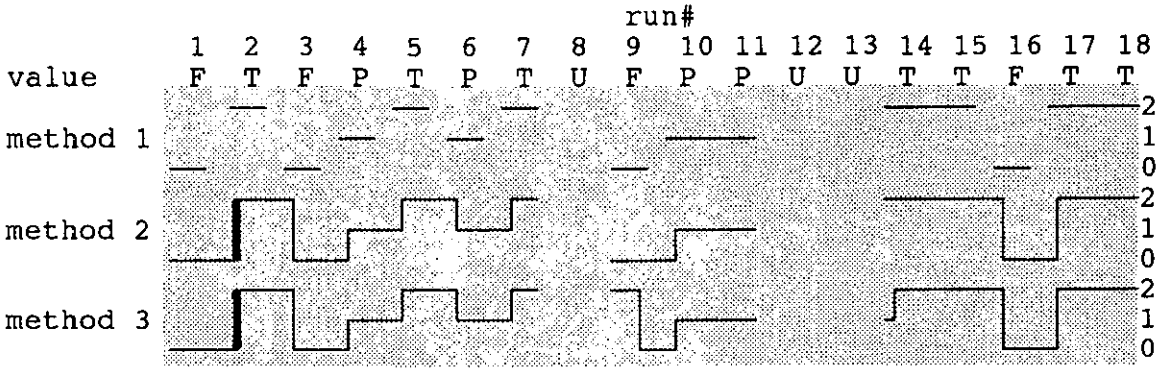


figure 5.4: Several examples of how to display the rule-values

The advantage of method 3 is that certain patterns are easily recognized (it looks more like a graph than the other two), but the disadvantage is that it is difficult to determine what value the rule has in a certain run (for example run #9:False). On the other side, this method remembers what the last value of the rule was before it became undefined, which can be useful if the value of run #1 is undefined and if this run is not the first run of the total process.
Method 1 has the disadvantage that if we want to display several rules besides each other, it becomes quite hard to determine what line belongs to what rule. Of course we will still have this problem even for method 3 if we have rules which are almost all the time undefined.

It is also possible to display the rule-values in just one or two character-lines but the problem then is, that the character-set is quite limited. For example method 4 (see figure 5.5) is an example of displaying the rule-values in two levels (level 0 = FAlse + POssible and level 1 = TRue + POssible).

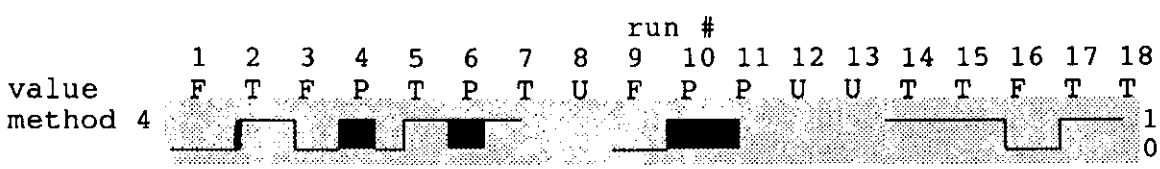


figure 5.5: Another example of how to display the rule-values

The advantage of using this method is that we can display more rules on the screen but the disadvantage is that the eyes get fixed first at the possible-values. The conclusion is that method 2 is the best way to display the rule-values of ASK, EVAL, MEMO and TEST rules.

Because every knowledge engineer wants to look at the process in a different way, we should provide a way to change the number of rules to display by the tracer (number of STATE rules (one level) and number of other rules (three levels)).

5.2.2 The Tree option

If we want to debug Expert systems or just verify their knowledge base, we also need an explanation system to provide the visibility of the rulebase.
For this option we are not so much interested in what value a certain rule has, but more how it obtained that value and what the effect of the rule on other rules is. This means that we need an explanation facility which will use retrospective reasoning (section 4.6).
All the different parts of a rule (see also section 3.4) tell us a little bit about the **structure of that rule** and how the rule will be used during the process:

ad A) the rule header:

The rule-name and the rule-text are two strings of determined length (in file rinfo.qqq) and they identify the rule. However we do not want to present a rule by its name or text, because these two string can be very long and they would require a lot of space in the tree. Therefore we will present a rule just by a unique number assigned to it in the file 'rinfo.qqq' (we will reserve three characters for every rule because we expect that a rulebase will contain at most 999 rules).

ad B) the rule-type:

This part tells us how the rule will be used during the process. Every rule-type needs a different presentation:
FACT rules obtain their value at the very beginning of the process and they never change during the process and so we just have to display that this rule is a FACT rule.
ASK rules obtain their value from asking a question and so we need only to display the question somewhere to 'explain' that this rule is an ASK rule. Displaying the question is the same as showing the rule-text string in a rule-information window.
TEST rules obtain their value by testing external data which will not be available anymore at the moment of debugging. Because most programmers are not interested in how the test looks, but more in that the rule is a TEST rule, we present these rules by their code. If the programmer is interested in how the test looks, he can find the code of the appropriate Pascal test in the file 'rtest.qqq'.
MEMO rules remember their values across runs and can only be changed by a THELSE of another rule. So for these rules the programmer will only be interested in when the rule was assigned the current value, which is the same as displaying the history of the rule at that moment.
STATE rules are assigned either by their initial value or by a state transition. So the programmer would want to know how the script section looks and what state transition changed the value of this state. A programmer mostly designs the process section by drawing a protocol network (for example a petri-network) and converting this network into state transitions. So the best way to explain the state transitions in which a

certain state is involved is to draw a protocol-like graph of that state.

Figure 5.6 contains not only the protocol network and the process section that State S1 is involved in, but also the protocol-like graph of that rule.

figure 5.6: An example of how to display STATE transitions

Notice that each State rule can be used in two different ways; In the TO list where it can become active (true) or in the FROM list where it can become inactive (false). Because mostly programmers are also interested in what triggers and what states are active, we will highlight all the rules in the state transitions which are true at that specific run.
EVAL rules need the values of other rules to obtain a certain value. The expressions of a rule can be very complex and difficult to oversee. The best way to explain the structure of these expressions to a programmer is to draw a graph like the explain facility FACE did [Hair,1988]. Only now we display the values of the used rules by just one character instead of giving colors to the rules (F=false, T=true, P=possible and U=undefined). Further, we draw the graph like the Inference Engine interprets it, so no smart reordering of the arcs and nodes (rule or expression).
Figure 5.7 is an example of an evaluation rule and its evaluation tree.

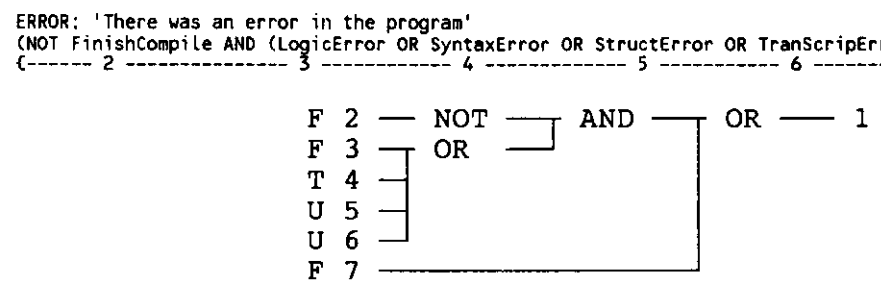


figure 5.7: An example of how to display an evaluation rule

Trigger rules are not a different kind of rules (they can be any kind of rule), but they need a different approach for their graphical representation. First of all a programmer wants to know if the rule is a trigger rule and secondly if it is, he wants to know in what state transitions this rule will be used. We will use the same method as the one we used to display the

STATE rules. Figure 5.8 shows us the graph for trigger Tr4 (12) of the example of figure 5.6.

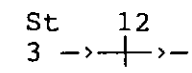


figure 5.8: An example of how to display a trigger rule

ad C) The INITIALLY section:

The initial value is only important at the beginning of the process. So we do not need to display this information in the graph itself but in a rule-information-window ("initially: true").

ad D) The THELSES section:

This section tells us what effect a rule can have on other rules. We distinguish two kinds of THELSES: Thelses-In and Thelses-Out.
If a rule has a Thelses-Out it means that this rule assigns a typical value to another rule under the condition that this rule has a certain value. If a rule has a Thelses-In it means that another rule has a Thelses-Out to this rule.
If we display the thelses in the evaluation tree of a rule, we can always determine during the simulation why certain rules are evaluated or what rules will also be evaluated during that run.

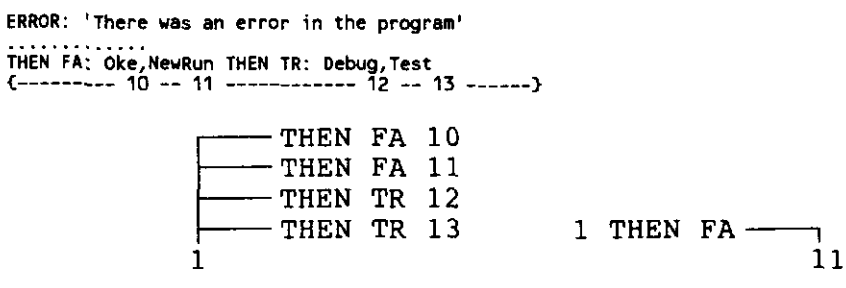


figure 5.9: An example of how to display the THELSES of a rule

For example the graph of rule 'Error' (1) tells us that if this rule evaluates to true then rule 10 and 11 will be set false and rule 12 and 13 will be set true (see figure 5.9).

Remarks:
-A rule can only have THELSE-DO's out.
-STATE rules cannot have Thelses-In at all and THELSE-GOAL's in are not possible for FACT and MEMO rules.
-In the last version of Simplexys the Rule Compiler generates an array which contains all the THELSES-in except the 'THEN GOAL's. These kind of THELSES will therefore not be displayed in the evaluation tree (only in the THELSES-out section).

So, in general, a tree of a rule will have the format of figure 5.10. In this figure every section is optional and depends on the kind of rule.

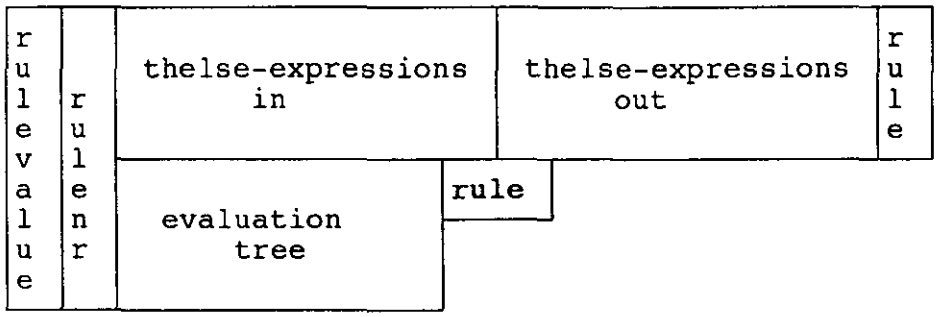
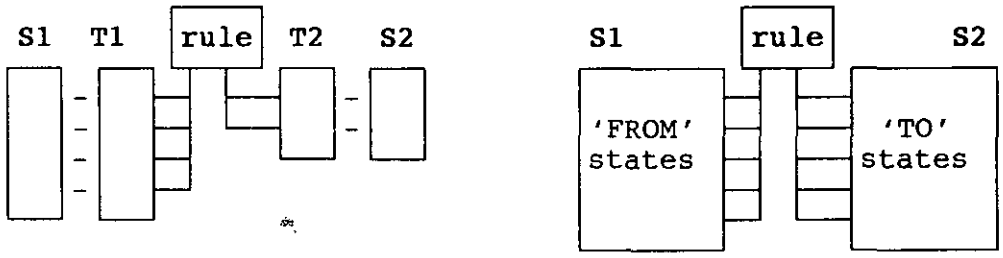


figure 5.10: Schematic diagram of how to represent a rule

As we saw already, STATE rules and rules used as triggers describe a total different part of the process (dynamic part) and therefore need a different presentation (see figure 5.11). Most programmers will not always be interested in how this section looks and so we must see the drawing of these trees as extra features which can be activated by stroking a special function key.



ON T1 FROM S1 TO rule
ON T2 FROM rule TO S2

ON rule FROM S1 TO S2

figure 5.11: Schematic diagram of how to present trigger and state rules

Remark: Because the structure of evaluation rules can be very complex and large, it is possible that the tree does not fit into the predefined window. If we still want to see the whole tree we need to scroll the display, and thus a Scrllock function.

5.3 Special Considerations

As discussed before, most kinds of syntax and semantic errors will be detected by the Rule Compiler or one of the checkers (Protocol Checker or Semantic Checker). But, it is possible that the rulebase becomes so big that it is impossible to check this rulebase for, for example, the semantic errors (this is due to current limitations of the checkers, which cannot handle

knowledge bases containing 200 or more rules).
The debugger however will still function for rulebases consisting of up to 999 different rules, because the additional memory needed for the debugger can be kept small by making the number of runs in one simulation block smaller. With the help of the built-in explanation facility, it should be quite easy then to recognize for example the semantic errors which would otherwise be detected by the semantic checker. Section 4.2 shows us the graphics of how these errors will appear in the evaluation trees drawn by the tree mode.
Some of the errors detected by the Protocol Checker can be recognized easily by the debugger (like 'selfloop' or 'net parts not connected') but most of the topological and dynamic errors cannot be detected clearly. They mostly result in some kind of strange behavior of the expert system during real-time execution.

Chapter 6: Ergonomics, the user-interface

6.1 Ergonomics in general

Because we already discussed what we need to display for debugging, we will now discuss some ergonomic aspects of how to organize the total display and control of the system.
Ergonomics seeks to maximize safety, efficiency and comfort by shaping the machine to the operator's capabilities. By linking the machine to the operator in this way a relationship is established where the machine presents information (displays) to the operator via the operator's sensory apparatus to which a response may be made by the operator to alter the machine's state (controls). The theory about ergonomics in general and what kind of displays and controls exist, is already examined by other researchers [Hoogendoorn,1989; Oborne,1987; Hendler,1988].
The design of interfaces for expert systems differs in a way from the design of 'normal' programs because we distinguish different users with different needs.
The user, who will directly communicate with the simulator/debugger will probably be a knowledge engineer or someone who wants to know more about the Simplexys rulebases. We expect that these users already know the basic ideas about Simplexys and so we can represent the important process-information in a knowledge-base-like way.

6.2 Visual displays

Visual displays are the most commonly used instruments for communicating information from the machine to the user.
As mentioned before, the first demands of the debugger were that it should be possible to display all the information in text mode on a monochrome monitor. This implies that the debugger is not allowed to use graphics and/or colors to display the process-information [1].
The debugger is a top-down display of the world of action of an expert system. By this we mean that on the top of the screen we display all the information which is most important to the total process: the rule-values of the State rules (tracer); on the bottom of the screen we display less important information like evaluation trees or rule-information windows (run information).
To emphasize certain information in a display without using graphics, we can highlight the characters (for example rule is true in evaluation tree) or inverse the 'colors' of the characters and their background (for example rule-name in tracer) or just use capitals (for example rule-name in rule-information window). By using one of these three possibilities we will not use extra space on the screen.
Figure 6.1 shows us a simplified layout of the total screen

---

[1] text mode, on an IBM PC, does include a number of "graphics characters"; these can be used of course

with its visual display windows.

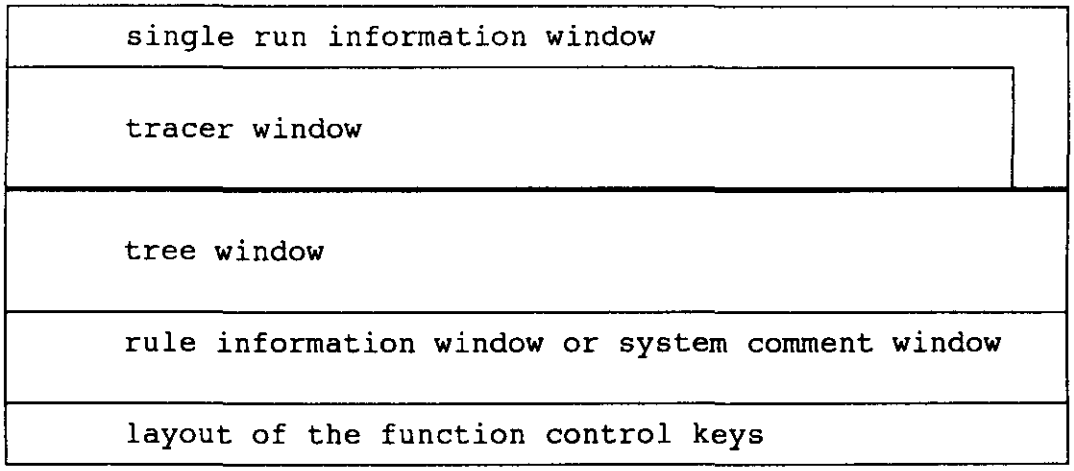


figure 6.1: The layout of the user interface

Notice that the two main visual displays, the tracer and the tree, are placed in the middle of the screen. These two main displays often consist of several smaller sub-displays (for each rule one).

6.2.1 Sizes of the displays

We let the user define the sizes of the two main displays because the number of rules that a user wants to display on the screen depends not only on how large his rulebase is but also on what this user prefers.
For large knowledge bases there must also be the possibility to define two separate virtual screens for both options (when the tracer is defined too large the debugger will switch to this mode automatically). Because rule- and run-information do not change extremely, the size of these (information) windows will be fixed during the whole debugging process.

6.2.2 Menus

If we let the user determine certain display options, we will need a multiple selection menu, where we can choose certain debugger options.
Because the total debugger consists of a simulator and a tracer, we also need a main menu where we can choose between simulation, examining or just changing the display options. After quitting one of these options, the debugger will always return to this main menu.

6.3 Controls

Controls represent the second link in the man-machine closed loop system and are very much the complement of displays because they allow the operator to return information to the system environment.
All the control keys should allow the user to communicate with

the machine and therefore they should be represented by single push buttons. For both options these function keys should have a similar function. This implies that some keys will have a meaning in one mode but no meaning in the other.

6.3.1 Situation of the controls

Because most of the users of the debugger will have some experience with the Pascal-packet 'Turbo Pascal' version 4 or 5, the layout of the control keys looks like the ones of this tool. This means that we will display the active keys in a horizontal row on the bottom of the screen.
The difference with Turbo Pascal is that for our two tools we only need to display seven different active function keys.

6.3.2 Cursor movement keys

All the needed cursor movements in the tracer and tree mode can be done with the four arrow keys '→','←','↑' and '↓' on the keyboard. Because the allowed cursor movements in both modes are quite limited, we will not use special computer input control devices like touch displays, light pens, bar-code scanners or mouses. Another reason not to use these devices was that the working space of the programmer will be quite limited and there will be no space left for special input devices like mouses. For example in the case of the blood-pressure controller the knowledge engineer will also use a blood-pressure-simulator beside the debugger which will show the real-time blood-pressure-signals on the screen again.

6.3.3 Feedback after a command

After pressing one of the function keys most of these commands will respond by changing a part of the screen-layout. Only the function keys 'STATE' and 'Scrll' are commands which only put the tree mode in a special mode. The result of these commands will therefore only be noticed after displaying another rule evaluation tree. To show to the user that these special options are activated we will reverse the 'colors' of the appropriate control-button on the bottom of the screen.

6.4 Special input

For more experienced users we support special keys for making shortcuts in the debugging process. We can reduce the number of interactions by defining special keys for moving to the first or last run of the simulation block.
Another helpful option can be not to display the extended rule-information window.
Displaying all the values of all the rules on the screen is not desirable and not possible (the knowledge base is usually too large). Therefore the system asks the user to type in the rule-number of the rule he wants to examine. Every user can look up in the appropriate file rinfo.qqq what rule-number belongs to what rule-name.
If the input is not the right format or is not within the

predefined limits, the system will generate a (soft-toned) beep and ask the question again. Informative feedback is given by showing the extended rule-information about the asked rule.

6.5 Examples of debugging

The best way to show how the total layout of the debugger looks, is by giving some screen-dump-examples of the debugger in operation.

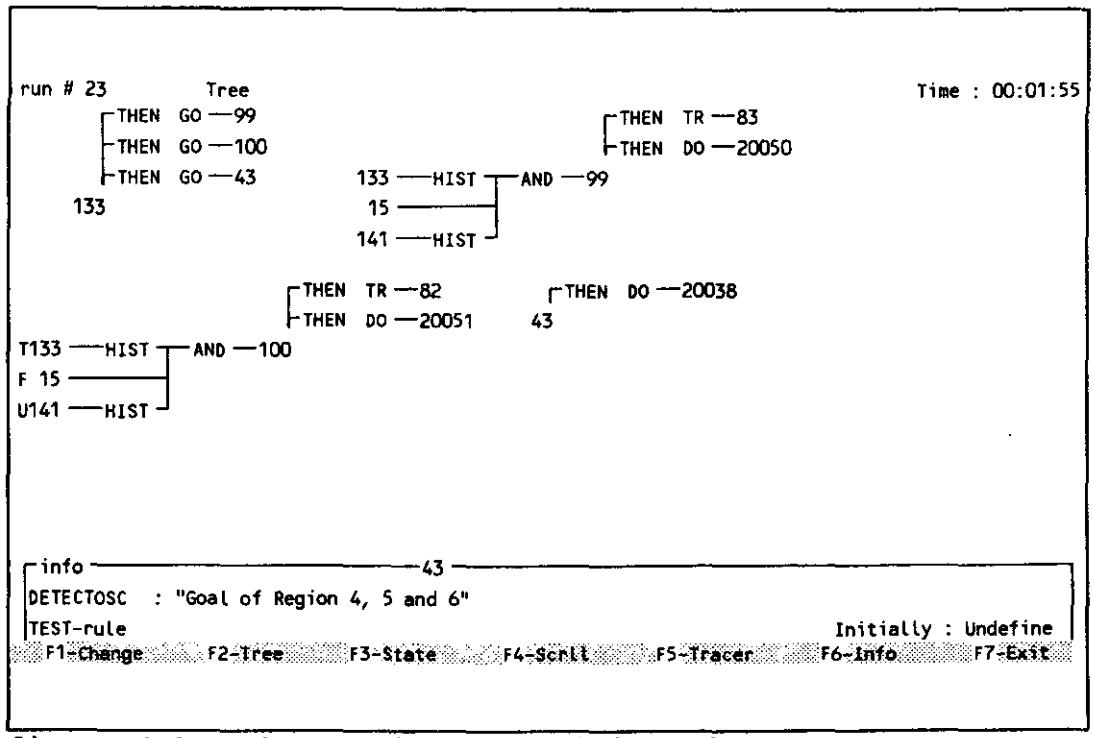


figure 6.2: Debugger in the explain-mode

For this example the debugger was installed for two virtual screens, one for the tracer (see figure 6.3) and one for the explain facility called tree (figure 6.2).

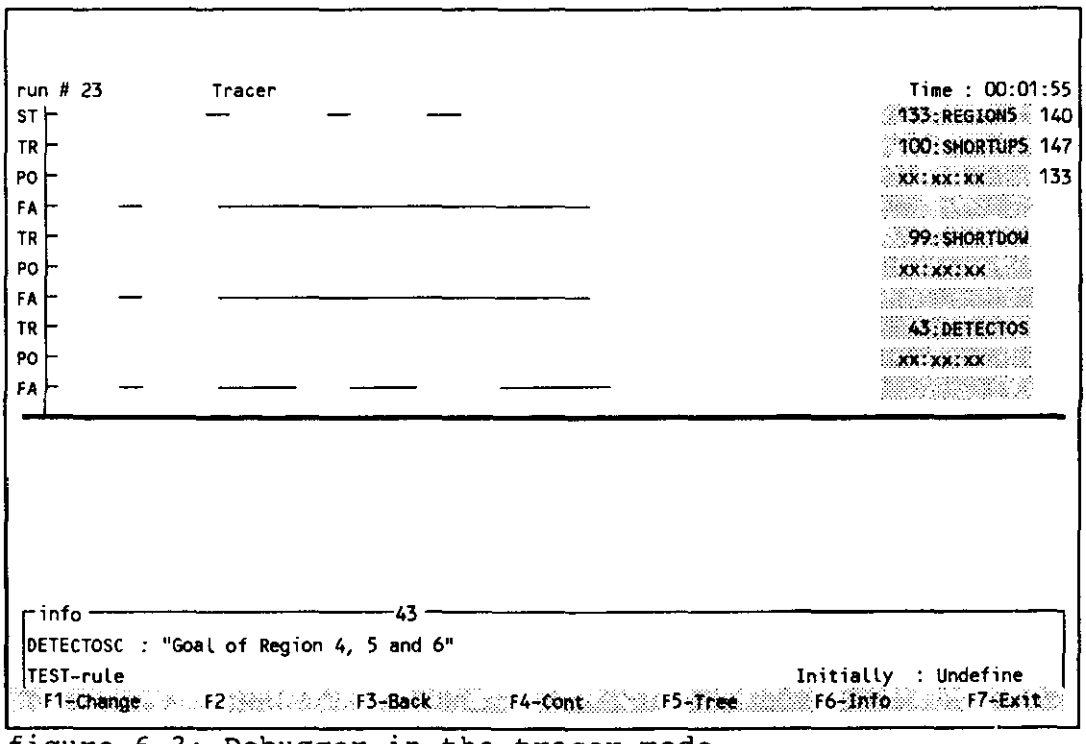


figure 6.3: Debugger in the tracer-mode

6.6 Conclusions

The final system turned out to be easy to use. The time needed to learn how to use the commands is very short, because almost all the possible commands are displayed on the bottom of the screen (no need to remember) and the appropriate actions belonging to a function key are carried out immediately after stroking the key (speed of performance).
All the commands are reversible except the read-next-simulation block command because it is not possible to go back in time (because of rule-histories).
Depending on the kind of user some aspects of the system will be used more than others. The Scrllock function key probably will not be used very much. A programmer can avoid using this special option key by installing the displays of the debugging tools in a convenient way.

Chapter 7: User's manual for the debugger

If an expert system is generated by the adapted SIMPLEXYS and executed in a real-time environment, then we can simulate the whole process again and examine it more carefully with the debugging tools. The simulator needs only the files rinfo.qqq,rhist.qqq and the dumpfile. Using the debugging tools in the simulation of a real-time process is following a predefined debugging process:

1. Initialization
2. choosing item in main menu:
   2.1. changing options
   2.2. debugging (tracing/explaining)
   2.3. simulating next block (until last block)
   2.4. exit

Each step needs a different approach and will therefore be discussed separately.

7.1 The initialization

If we execute the simulator it will first print the header of the Simplexys application:

```
     A SIMPLEXYS EXPERT SYSTEM APPLICATION

      Copyright (C) 1987-89, Hans Blom
Eindhoven University of Technology, Netherlands
             All rights reserved

Simulating block of 100 runs
```

At this point the simulator starts simulating the first block. After simulating 100 runs (default size for a simulation block) it will show the message how many runs it simulated and what the start- and finish-time of that block are. We also get the main menu on the screen now.

7.2 The main menu

The main menu consists of four possibilities (see figure 7.1).

```
Menu :
------
1 : Change Options
2 : Run Debugger
3 : Next Block
4 : Exit
```

figure 7.1: The layout of the main menu

Each option can be selected by just pressing the key with the appropriate number ('1','2','3' or '4') or by pressing the appropriate function key ('F1','F2','F3' or 'F4') or just by pressing the key with the first character of the explanation-string of the action ('O','o','R','r','N','n','E' or 'e').

7.3 Changing options

The first option of the main menu gives the possibility to change the start-time of the next simulation block and the possibility to change some display options (see figure 7.2).

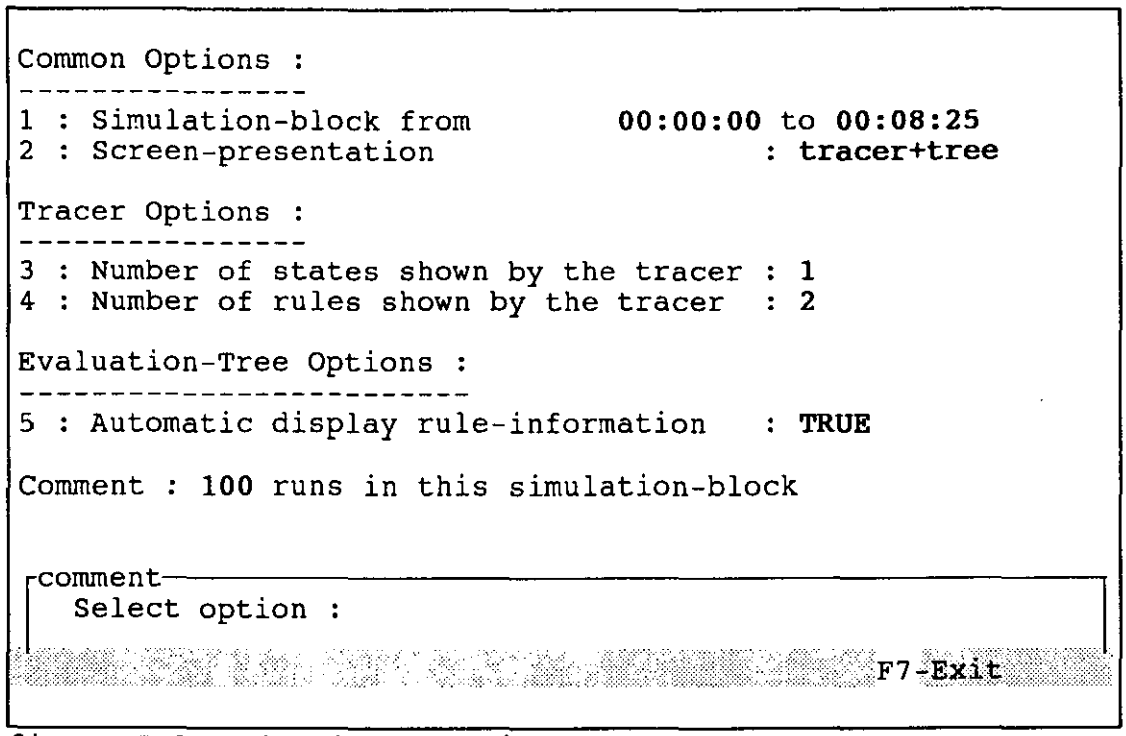


figure 7.2 : The change-option menu

Each option can be chosen by just pressing the appropriate key ('1' to '5') or function key ('F1' to 'F5'). For option 3,4 and 5 the cursor will move to the default value and the user may type in the value. The computer will check if that value is within the predefined bounds.
For option 1 the user has to enter a string containing 8 characters. If he enters a shorter string, the start-time will not be changed.
Option 2 is special because there are only three allowed values. The appropriate key will automatically 'switch' the value. Choosing for the option 'tree' will allow the user to use the debugger together with a small start-up program ("explain.pas") as a stand alone explanation facility like FACE was.
By pressing function key F7, the character 'e' or 'E' or the key combination Ctrl-Z will quit this mode and return to the main menu. If one of the options 2 to 5 is changed, then these

options will keep this value during the whole debugging process (option 1 is block-dependent).

7.4 The debugger

Option two of the main menu allows to examine the last simulated block of the process more carefully with the two debugging tools: tracer and tree. Depending on how the user defined the tracer, the layout of the two tools can look very different but the default one is shown in figure 7.3 (in this case the tracer option is installed for one State rule-representation (1) and two 'normal' rule-representation (11 and 111)).

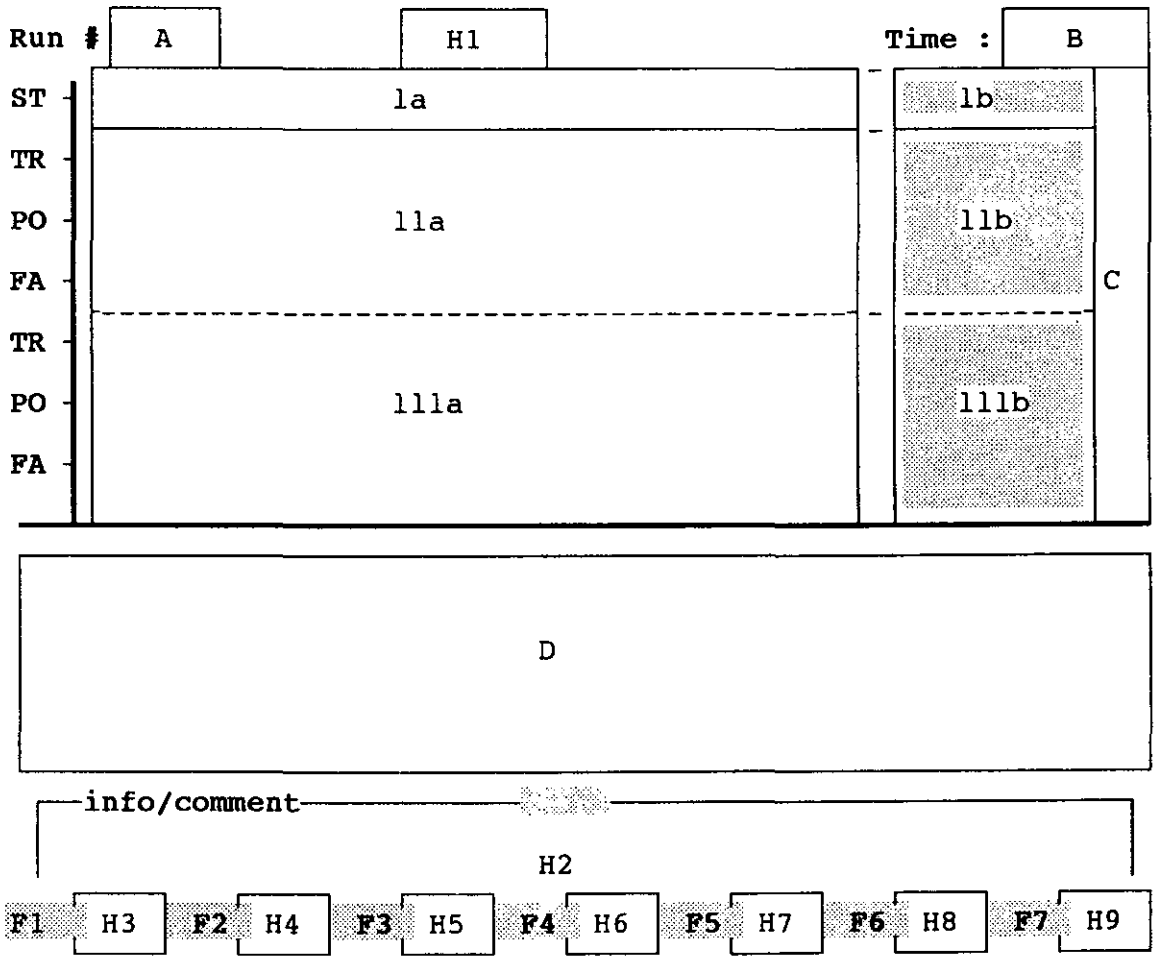


figure 7.3: The screen-layout of the debugging tools

The total screen is divided into three types of independent windows:
A-› Tracer or process information windows 1 to 111
B-› Single run information windows A to D
C-› help information windows H1 to H9

7.4.1 Tracer or process information windows

The windows 1a,11a and 111a contain the graphical presentations of the run-values of the three chosen rules and the windows 1b,11b and 111b are the information windows of these same rules. Notice that the normal rule information windows (11b and 111b) are larger than the State information window (1b) and so we can also display the history of the rule which it had at the first run.

7.4.2 Single run information windows

There are four run information windows:

-› Window A
  A tells us which run we are examining with the tree mode. The information in this window is only changing when we are moving the cursor to the left or to the right in the tracer mode.

-› Window B
  B tells us when the run actually took place. For real-time processes this window shows us the real-time of the run and for simulated processes this window shows us how many seconds are passed since the process began.

-› Window C
  C tells us what State(s) are active (true) in this run.

-› Window D or the tree option window
  This window shows us the structure of one or more rules and highlights them if they are true at that run. Also the values of the leaves of the structured tree are displayed (T=true, F=false, P=possible, U=undefined).

7.4.3 Help information windows

There are nine different help information windows:

-› Window H1
  H1 shows us what mode is active (tracer or tree) at that moment. We need to display what mode we are in because for some rules like FACT rules only the information window H2 is displayed and we do not know then what mode is active at that moment.

-› Window H2
  H2 is a more complete information window than 1b,11b or 111b. It contains the rule-number, the rule-name, the rule-string, the kind of rule (FACT, ASK, TEST, MEMO, STATE) and the initial value. When the rule is also used as a trigger then this extra information is added to the information window ("used as trigger").

-› The windows H3 t/m H9
  These windows contain the action-strings belonging to the appropriate function keys. Some of these function keys contain the same string for both options:
    F1-Change : With this function key we can display in the tracer option a different rule from the one in the window where the cursor is. In the tree option this function key will clear window D and it will display the structured tree of the entered new rule-number.

F6-Info : With this function key we can display the extended information of every rule where the cursor is pointing, in window H2. So in the tracer mode we will get the information of the rule displayed in the windows 1a,11a or 111a, which is a more complete rule information window than window 1b,11b or 111b.
In the tree mode it is possible to move the cursor to the rule of which we want more information.
If the user presses this key twice, he will be able to see extended information of all the rules in the rulebase (starts with the rule first entered in the tree mode). By pressing any key but the F6-key the user will quit this special feature and go back to the cursor position where he started this special feature.
F7-Exit also Ctrl-Z: With this function key we can leave the tracer and/or tree mode and return to the main menu for reading a new simulation block or to quit the tool. If there is no simulation block left (number of runs in last block is smaller than 100) we will also automatically quit the program.

Function keys which differ for both options:

Special Tracer option keys:

F3-Back and F4-Cont : It is not possible to show 100 runs at once at the screen and so we need special keys to walk fast through these 100 runs. With the key F4 it is possible to move one half window forward (default value is 29 runs) and with F3 we can move the same number of runs back again.
F5-Tree : With this function key we can switch to the tree mode: the cursor will move to window D.

Special Tree option keys

F2-Tree : This function key allows us to display the trees of the primitives of an already displayed structured tree.
F3-State and F4-Scrll : These function keys can activate special features for the tree mode. When the State feature is active, all the State transitions which use this rule are displayed in a special way (see section 5.2.2). This feature can be useful when the user wants to have a look at the process description or if he wants to know what states will be active in the next run. When the Scrll feature is active all the structured trees will be displayed line by line. This feature can be useful if the structure of one rule is so large that it does not fit in any window and we still want to look at all its leaves. When a feature is active, the appropriate string will be displayed in reverse (window H5 and/or H6).
F5-Tracer: This function key allows the user to switch back to the tracer mode (window 11a).

7.4.4 Cursor movement keys

With the exception of the special Tracer-keys F3 and F4, the cursor movement keys are not implemented in the option key-menu.

For the tracer mode (or when the cursor is in the window 1a,11a or 111a):

left arrow ('←'): With this key we can move back in time: current_run:=current_run-1. When the cursor reaches the left end of the window, all the graphs will move to the right. This is possible until we reached the first run of the simulation block (run #1).

right arrow ('→'): With this key we can move forward in time: current_run:=current_run+1. When the cursor reaches the right end of the window all the graphs will move one run to the left. This is possible until we reache the last run of the simulation block (for most of the simulation blocks: run #100).

Home ('Home'): This key will move the cursor to the beginning of the window 1a,11a or 111a.

End ('End'): This key will move the cursor to the end of the window 1a,11a or 111a.

For the tree mode (or when the cursor is in window D):

Right Arrow ('→'): With this key we can move to the position of the rule-numbers of the THELSES-expressions-out of the current 'active' tree.

Left Arrow ('←'): With this key we can move back to the left side of the structured tree.

With these two keys it is possible to get to the position in the screen of all the rule-numbers used in the tree.

With function key F2 it is possible to draw a new sub-tree.

Home ('Home'): This key will redraw the socalled parent structured tree. So together with function key F2 and the four arrow keys we can show all the trees of the leaves of the current parent tree.

7.5 Simulating next block

Option three of the main menu will clear the screen and try to simulate another block of 100 runs. The simulation will start at the time defined in the 'Change Options' menu (default is the finish-time of the last simulated block).

7.6 Exit the debugger

The debugger will not simulate the process to the end, but will quit the process by just making all the states inactive.

7.7 Explain

Besides examining the expert system by simulating the process, it is also possible just to examine the structures of the rulebase with the explanation facility 'tree'. To do so, there is developed a small program called 'explain' which includes the tree option of the debugger in its program.

Chapter 8: Conclusions

The simulator-debugger combination can be used by all kinds of expert system users, but the knowledge engineers who know a little bit about Simplexys will have more profit from these facilities; users who know more about the problem-field are more able to optimize their knowledge-bases. The debugging tools can help these system-designers in doing this more efficiently and faster.

After trying out the new Inference Engine with the two debugging tools on some small rulebases, there was a great need of examining the large rulebase of the blood-pressure controller. Although this rulebase contained many more states and although it was used for complex real-time processes, containing many links to Pascal, the new Inference Engine and Simulator had no problems on debugging/examining it. Lammers, however, used the debugger/simulator mainly for examining the process more carefully. He tried to find out why and when certain rules were evaluated during the real-time process. This examination of the process resulted in several modifications of the rulebase.

Experts and knowledge engineers will not trust an existing expert system by just looking at the final decisions it takes. They want to know how the system came to these decisions. The debugger can be a tool to convince these people of the efficiency and correctness of the expert system.

We have to mention here that it saves a lot of time in debugging if the programmer makes notes during the real-time process of the times when certain 'strange' decisions were taken (the special time-change option in the "Change option"-menu turned out to be very useful).

References

Aleksander, I. et al.
ROBOT TECHNOLOGY. Vol. 6: Decision and intelligence.
London: Kogan Page, 1986.

Blom, J.A.
THE SIMPLEXYS EXPERT SYSTEMS TOOLBOX: Simplexys manual.
Internal document.
Division of Medical Electrical Engineering, Faculty of Electrical Engineering, Eindhoven University of Technology, June 1988.

Boon, P.M.G.
EFFICIENTIE EN CORRECTHEID VAN SIMPLEXYS EXPERT SYSTEMS.
M.Sc. Thesis. Division of Medical Electrical Engineering, Faculty of Electrical Engineering, Eindhoven University of Technology, 1987.

Brown, A.R. and W.A. Sampson
PROGRAM DEBUGGING: The prevention and cure of program errors.
London: Macdonald/New York: American Elsevier, 1973.
Computer monographs, Vol. 18.

Cassel, D.
THE STRUCTURED ALTERNATIVE: Program design, style, and debugging.
Reston, Virginia: Reston Publishing Co., 1983.

Hair, P.J.A. de
REALISATIE VAN EEN UITLEGFACILITEIT VOOR SIMPLEXYS EXPERT-SYSTEMEN.
M.Sc. Thesis. Division of Medical Electrical Engineering, Faculty of Electrical Engineering, Eindhoven University of Technology, 1988.

Hasling, D.W. et al.
STRATEGIC EXPLANATIONS FOR A DIAGNOSTIC CONSULTATION SYSTEM.
Int. J. Man-Mach. Stud., Vol. 20, No. 1(1984), p. 3-19. Reprinted in: Developments in Expert Systems. Ed. by M.J. Coombs.
London: Academic Press, 1984. Computers and people series. P. 117-133.

Harmon, P. and D. King
EXPERT SYSTEMS: Artificial intelligence in business.
Chichester: Wiley, 1985.

Hausen, H.-L. (ed.)
SOFTWARE VALIDATION: Inspection, testing, verification, alternatives.
Proc. Symp., Darmstadt, 25-30 Sept. 1983.
Amsterdam: North-Holland, 1984.

Hendler, J.A. (ed.)
EXPERT SYSTEMS: The user interface.
Norwood, N.J.: Ablex, 1988. Human-computer interaction: a series of monographs, edited volumes and texts.

Hoogendoorn, P.
THE DESIGN OF A RULE BASED BLOOD PRESSURE CONTROLLER.
M.Sc. Thesis. Division of Medical Electrical Engineering, Faculty of Electrical Engineering, Eindhoven University of Technology, 1989.

Khoroshevsky, V.F.
ATN-BASED EXPLANATION SUBSYSTEMS: Design and implementation.
Comput. & Artif. Intell. (Czechoslovakia), Vol. 4(1985), p. 289-311.

Neches, R. et al.
ENHANCED MAINTENANCE AND EXPLANATION OF EXPERT SYSTEMS THROUGH EXPLICIT MODELS OF THE DEVELOPMENT.
In: Proc. IEEE Workshop on Principles of Knowledge-Based Systems, Denver, Col., 3-4 Dec. 1984. New York: IEEE, 1984. P. 173-183.
Reprinted in: Principles of Expert Systems. Ed. by A. Gupta and B.E. Prasad. New York: IEEE Press, 1988. IEEE Press selected reprint series. P. 283-293.

Oborne, D.J. (ed.)
ERGONOMICS AT WORK. 2nd ed.
New York: Wiley, 1987.

Osterweil, L.
INTEGRATING THE TESTING, ANALYSIS AND DEBUGGING OF PROGRAMS.
In: Software Validation: Inspection, testing, verification, alternatives. Proc. Symp., Darmstadt, 25-30 Sept. 1983. Ed. by H.-L. Hausen.
Amsterdam: North-Holland, 1984. P. 73-102.

Pau, L.F.
PROTOTYPING, VALIDATION AND MAINTENANCE OF KNOWLEDGE BASED SYSTEMS SOFTWARE.
In: Proc. 3rd Annual Expert Systems in Government Conf., Washington, 19-23 Oct. 1987. Ed. by H.J. Antonisse.
New York: IEEE, 1987. P. 248-253.

Reisig, W.
PETRI NETS: An introduction.
Berlin: Springer, 1985. EATCS: Monographs on theoretical computer science, Vol. 4.

Rolston, D.W.
PRINCIPLES OF ARTIFICIAL INTELLIGENCE AND EXPERT SYSTEMS DEVELOPMENT.
New York: McGraw-Hill, 1988.

Seviora, R.E.
KNOWLEDGE-BASED PROGRAM DEBUGGING SYSTEMS.
IEEE Software, Vol. 4, No. 3(May 1987), p. 20-32.

Shortliffe, E.H.
COMPUTER-BASED MEDICAL CONSULTATIONS: MYCIN.
New York: American Elsevier, 1976.
Elsevier computer science library: Artificial intelligence series, Vol. 2.

Smith, T.
SECRETS OF SOFTWARE DEBUGGING.
Blue Ridge Summit, Penn.: TAB Books, 1984.

Tassel, D. van
PROGRAM STYLE, DESIGN, EFFICIENCY, DEBUGGING, AND TESTING.
Englewood Cliffs, N.J.: Prentice-Hall, 1974.

Enclosure 1: Possible modes of the Inference Engine

The Inference Engine can be used in three different modes (see section 5.1). To install the Inference Engine in one of these modes, you have to define one or two conditional compilation symbols [1]:

1- Inference Engine without generating a dumpfile
   no extra conditional symbols
2- Inference Engine with generating the dumpfile 'simplex.sav'
   define conditional symbol EXAMEN
3- Inference Engine as simulator/debugger
   define conditional symbols EXAMEN and SIMUL

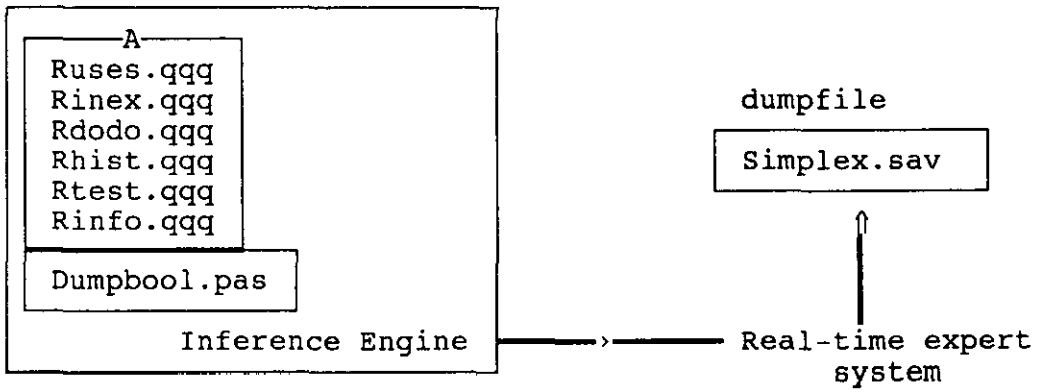


figure 1: The adjusted Inference Engine

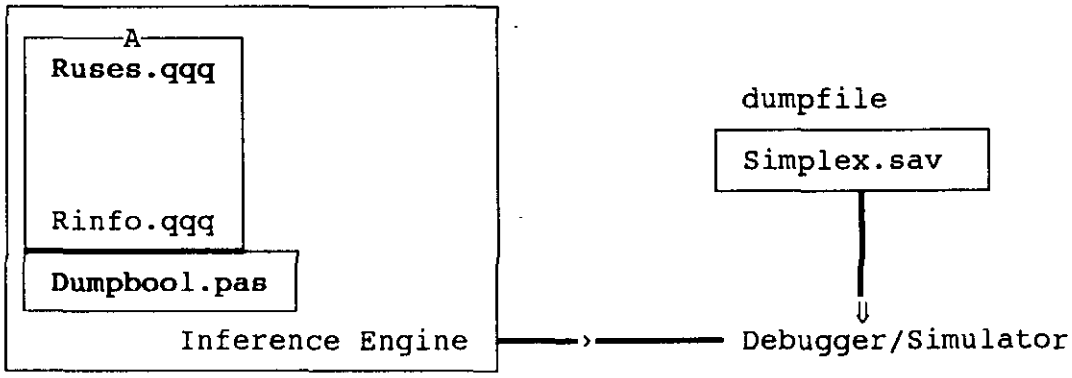


figure 2: The Inference Engine as Simulator

So the Inference Engine in mode two will generate an expert system (see figure 1), which can be simulated by the Inference Engine in mode three (see figure 2).
All the files in block A in figure 1 and 2 are generated by the Simplexys Rule Compiler. The files in figure 2, which are

[1] The procedure to do this more conveniently will be incorporated into the Options Builder tool.

represented bold, are different for both modes.

Enclosure 2: Adjustments for the Inference Engine

Procedure	mode 1 Old Inf. Eng.	mode 2 New Inf. Eng.	mode 3 Simulation
init_time	_time0:=sys_time or 0	_time0:=sys_time or 0 dump_time(_time0)	_time0:= read_time
update_time	_time:=.....	_time:=..... dump_time(_time)	_time:= read_time
eval_rule	_ask : value:= ASKval(...)	_ask : value:= ASKval(...) dump_rule(value)	_ask:value:= read_rule
	_test : value:= _FTEST(...)	_test : value:= _FTEST(...) dump_rule(value)	_test:value:= read_rule
_applyHIST	dummy:= _FHIS(...)	dummy:=_FHIS(...) if dummy then value:=TR else value:=FA dump_rule(value)	value:= read_rule if value=TR dummy:=true else dummy:=false _his_arr[]:= u
_getFACTS	_R[rule]:= ASKval(...)	_R[rule]:= ASKval(...) dump_rule(value)	R[rule]:= read_rule
main program	infer	open_dumpfile infer close_dumpfile	open_dumpfile infer close_dumpfile

Enclosure 3: dumpbool.pas

Depending on whether the conditional symbol SIMUL is active or not, the debugger-file dumpbool.pas will look quite differently.
Dumpbool.pas will be included by the Inference Engine and contains all the procedures and functions for writing/reading values to/from the dumpfile 'simplex.sav'.
For the Inference Engine in mode three, this file will also include all the procedures and functions needed for the debugger (see table 1).

Table I : The procedures of file dumpbool.pas

not SIMUL	SIMUL
	debug
dump_buf	read_buf
dump_time	read_time
dump_ rule	read_rule
	find_history
	{ counts the number of history-expressions in rhist.qqq. }
open_dumpfile	open_dumpfile
close_dumpfile	close_dumpfile
	expand
	make_arr
	{ puts the result of the history-expressions and all the rule-values into two big trace-arrays }
	{ stops the process at a certain point for examining the last block }

Enclosure 4: The debugging procedure

The file debug.pas, which will be included by dumpbool.pas, is actually only one procedure: procedure 'debug'. This procedure contains all that is needed to display and use the tracer and the explanation facility called tree (file debugmod.pas).
In order to trace, we have to make some large arrays during the simulation:

* for each run we need to remember the value of each rule
  (2 bits * [number of runs in one block] * [number of rules])
* for each run we need to remember the value of history expressions
  (2 bits * [number of runs in one block] * [number of expressions])
* for each run we need to remember the run-time of that run
  (long integer * [number of runs in one block])
* for the first run of the block, we need to remember the history of all the rules

The procedure debug consists of several internal procedures:

1. Debug
   - 1.1 show_time
     display the time (longinteger) as __:__:__
   - 1.2 Invers_color
     Changing color of text and background
   - 1.3 Screenwriter
     To display all the needed menus and to determine sizes of different visual displays
     - 1.3.1 Get_kind
     - 1.3.2 Get_init
   - 1.4 Check
     Check if input from user is within the predefined bounds
   - 1.5 graph1
     one-level method for displaying rule-values in tracer
   - 1.6 make_dyn_list
     three-level method for displaying rule-values in tracer
     - 1.6.1 drawdyn
   - 1.7 write_inv
     writing the rule-number + shortened rule-name
   - 1.8 write_info
     system asks for new rule-number to display
   - 1.9 what_hist
     how to display the history of a rule
   - 1.10 Change_color
     Inversing the colors of the control buttons on the screen
   - * 1.11 change_option
     possibility to change the five display options for tracer and tree
     - * 1.11.1 What_time
     - * 1.11.2 switch
   - * 1.12 Make_tree
     displaying tree structures of rules (=explain

```
        facility)
*       1.12.1 fin_draw
*            1.12.1.1 Showiftrue
*            1.12.1.2 Convert
*            1.12.1.3 count
*            1.12.1.4 HandleWindow
*            1.12.1.5 trigger_list
*            1.12.1.6 state_list
*            1.12.1.7 thelses
*                 1.12.1.7.1 Conv_thelses
*            1.12.1.8 DrawTree1
*                 1.12.1.8.1 Conv_code
*                 1.12.1.8.2 Make_con
*            1.12.1.9 DrawTree2
*       1.12.2 Find_rule
*       1.12.3 HandleMove
   1.13 conv_code
        transferring bool-values into numbers
   1.14 write_time
        display new run-time and new run-number
   1.15 get_graph
        displaying the rule-values of one rule in tracer mode
   1.16 move_cursor
        handling all possible key-strokes of user in tracer
        mode
        1.16.1 make_init
        1.16.2 update_graph
        1.16.3 flash_back
        1.16.4 continu
        1.16.5 follow
   1.17 rd_data
        build up the whole tracer: display rules until no
        sub-display is left free in the tracer mode
        1.17.1 get_states
```

The procedures/functions marked with a '*', are situated in the file debugmod.pas.
The special program 'explain' contains the same procedures/functions only some of them are 'empty' (the ones referring to the tracer).

Eindhoven University of Technology Research Reports
Faculty of Electrical Engineering

ISSN 0167-9708
Coden: TEUEDE

(205) Butterweck, H.J. and J.H.F. Ritzerfeld, M.J. Werter
FINITE WORDLENGTH EFFECTS IN DIGITAL FILTERS: A review.
EUT Report 88-E-205. 1988. ISBN 90-6144-205-2

(206) Bollen, M.H.J. and G.A.P. Jacobs
EXTENSIVE TESTING OF AN ALGORITHM FOR TRAVELLING-WAVE-BASED DIRECTIONAL DETECTION AND PHASE-SELECTION BY USING TWONFIL AND EMTP.
EUT Report 88-E-206. 1988. ISBN 90-6144-206-0

(207) Schuurman, W. and M.P.H. Weenink
STABILITY OF A TAYLOR-RELAXED CYLINDRICAL PLASMA SEPARATED FROM THE WALL BY A VACUUM LAYER.
EUT Report 88-E-207. 1988. ISBN 90-6144-207-9

(208) Lucassen, F.H.R. and H.H. van de Ven
A NOTATION CONVENTION IN RIGID ROBOT MODELLING.
EUT Report 88-E-208. 1988. ISBN 90-6144-208-7

(209) Jóźwiak, L.
MINIMAL REALIZATION OF SEQUENTIAL MACHINES: The method of maximal adjacencies.
EUT Report 88-E-209. 1988. ISBN 90-6144-209-5

(210) Lucassen, F.H.R. and H.H. van de Ven
OPTIMAL BODY FIXED COORDINATE SYSTEMS IN NEWTON/EULER MODELLING.
EUT Report 88-E-210. 1988. ISBN 90-6144-210-9

(211) Boom, A.J.J. van den
H_∞-CONTROL: An exploratory study.
EUT Report 88-E-211. 1988. ISBN 90-6144-211-7

(212) Zhu Yu-Cai
ON THE ROBUST STABILITY OF MIMO LINEAR FEEDBACK SYSTEMS.
EUT Report 88-E-212. 1988. ISBN 90-6144-212-5

(213) Zhu Yu-Cai, M.H. Driessen, A.A.H. Damen and P. Eykhoff
A NEW SCHEME FOR IDENTIFICATION AND CONTROL.
EUT Report 88-E-213. 1988. ISBN 90-6144-213-3

(214) Bollen, M.H.J. and G.A.P. Jacobs
IMPLEMENTATION OF AN ALGORITHM FOR TRAVELLING-WAVE-BASED DIRECTIONAL DETECTION.
EUT Report 89-E-214. 1989. ISBN 90-6144-214-1

(215) Hoeijmakers, M.J. en J.M. Vleeshouwers
EEN MODEL VAN DE SYNCHRONE MACHINE MET GELIJKRICHTER, GESCHIKT VOOR REGELDOELEINDEN.
EUT Report 89-E-215. 1989. ISBN 90-6144-215-X

(216) Pineda de Gyvez, J.
LASER: A LAyout Sensitivity ExploreR. Report and user's manual.
EUT Report 89-E-216. 1989. ISBN 90-6144-216-8

(217) Duarte, J.L.
MINAS: An algorithm for systematic state assignment of sequential machines - computational aspects and results.
EUT Report 89-E-217. 1989. ISBN 90-6144-217-6

(218) Kamp, M.M.J.L. van de
SOFTWARE SET-UP FOR DATA PROCESSING OF DEPOLARIZATION DUE TO RAIN AND ICE CRYSTALS IN THE OLYMPUS PROJECT.
EUT Report 89-E-218. 1989. ISBN 90-6144-218-4

(219) Koster, G.J.P. and L. Stok
FROM NETWORK TO ARTWORK: Automatic schematic diagram generation.
EUT Report 89-E-219. 1989. ISBN 90-6144-219-2

(220) Willems, F.M.J.
CONVERSES FOR WRITE-UNIDIRECTIONAL MEMORIES.
EUT Report 89-E-220. 1989. ISBN 90-6144-220-6

(221) Kalasek, V.K.I. and W.M.C. van den Heuvel
L-SWITCH: A PC-program for computing transient voltages and currents during switching off three-phase inductances.
EUT Report 89-E-221. 1989. ISBN 90-6144-221-4

Eindhoven University of Technology Research Reports
Faculty of Electrical Engineering

ISSN 0167-9708
Coden: TEUEDE

(222) Jóźwiak, L.
THE FULL-DECOMPOSITION OF SEQUENTIAL MACHINES WITH THE SEPARATE REALIZATION OF THE NEXT-STATE AND OUTPUT FUNCTIONS.
EUT Report 89-E-222. 1989. ISBN 90-6144-222-2

(223) Jóźwiak, L.
THE BIT FULL-DECOMPOSITION OF SEQUENTIAL MACHINES.
EUT Report 89-E-223. 1989. ISBN 90-6144-223-0

(224) Book of abstracts of the first Benelux-Japan Workshop on Information and Communication Theory, Eindhoven, The Netherlands, 3-5 September 1989.
Ed. by Han Vinck.
EUT Report 89-E-224. 1989. ISBN 90-6144-224-9

(225) Hoeijmakers, M.J.
A POSSIBILITY TO INCORPORATE SATURATION IN THE SIMPLE, GLOBAL MODEL OF A SYNCHRONOUS MACHINE WITH RECTIFIER.
EUT Report 89-E-225. 1989. ISBN 90-6144-225-7

(226) Dahiya, R.P. and E.M. van Veldhuizen, W.R. Rutgers, L.H.Th. Rietjens
EXPERIMENTS ON INITIAL BEHAVIOUR OF CORONA GENERATED WITH ELECTRICAL PULSES SUPERIMPOSED ON DC BIAS.
EUT Report 89-E-226. 1989. ISBN 90-6144-226-5

(227) Bastings, R.H.A.
TOWARD THE DEVELOPMENT OF AN INTELLIGENT ALARM SYSTEM IN ANESTHESIA.
EUT Report 89-E-227. 1989. ISBN 90-6144-227-3

(228) Hekker, J.J.
COMPUTER ANIMATED GRAPHICS AS A TEACHING TOOL FOR THE ANESTHESIA MACHINE SIMULATOR.
EUT Report 89-E-228. 1989. ISBN 90-6144-228-1

(229) Oostrom, J.H.M. van
INTELLIGENT ALARMS IN ANESTHESIA: An implementation.
EUT Report 89-E-229. 1989. ISBN 90-6144-229-X

(230) Winter, M.R.M.
DESIGN OF A UNIVERSAL PROTOCOL SUBSYSTEM ARCHITECTURE: Specification of functions and services.
EUT Report 89-E-230. 1989. ISBN 90-6144-230-3

(231) Schemmann, M.F.C. and H.C. Heyker, J.J.M. Kwaspen, Th.G. van de Roer
MOUNTING AND DC TO 18 GHz CHARACTERISATION OF DOUBLE BARRIER RESONANT TUNNELING DEVICES.
EUT Report 89-E-231. 1989. ISBN 90-6144-231-1

(232) Sarma, A.D. and M.H.A.J. Herben
DATA ACQUISITION AND SIGNAL PROCESSING/ANALYSIS OF SCINTILLATION EVENTS FOR THE OLYMPUS PROPAGATION EXPERIMENT.
EUT Report 89-E-232. 1989. ISBN 90-6144-232-X

(233) Nederstigt, J.A.
DESIGN AND IMPLEMENTATION OF A SECOND PROTOTYPE OF THE INTELLIGENT ALARM SYSTEM IN ANESTHESIA.
EUT Report 90-E-233. 1990. ISBN 90-6144-233-8

(234) Philippens, E.H.J.
DESIGNING DEBUGGING TOOLS FOR SIMPLEXYS EXPERT SYSTEMS.
EUT Report 90-E-234. 1990. ISBN 90-6144-234-6

(235) Heffels, J.J.M.
A PATIENT SIMULATOR FOR ANESTHESIA TRAINING: A mechanical lung model and a physiological software model.
EUT Report 90-E-235. 1990. ISBN 90-6144-235-4