Knowledge base correctness checking for SIMPLEXYS expert systems

tu/e
Research Report
ISSN 0167-9708
Coden: TEUEDE

Eindhoven
University of Technology
Netherlands

Faculty of Electrical Engineering

Knowledge Base Correctness Checking for SIMPLEXYS Expert Systems

by
J.M.A. Lutgens

EUT Report 90-E-240
ISBN 90-6144-240-0

May 1990

Eindhoven University of Technology Research Reports

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
Eindhoven The Netherlands

ISSN 0167- 9708 Coden: TEUEDE

KNOWLEDGE BASE CORRECTNESS CHECKING FOR SIMPLEXYS EXPERT SYSTEMS

by

J.M.A. Lutgens

EUT Report 90-E-240
ISBN 90-6144-240-0

Eindhoven
May 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 October 1989 until May 1990 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, Ph.D.

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Lutgens, J.M.A.

Knowledge base correctness checking for Simplexys expert systems / by J.M.A. Lutgens. - Eindhoven: Eindhoven University of Technology, Faculty of Electrical Engineering. - Fig., tab. - (EUT report, ISSN 0167-9708, 90-E-240)
Met lit. opg., reg.
ISBN 90-6144-240-0
SISO 608.1 UDC 616-089.5 NUGI 742
Trefw.: anesthesie; patiëntbewaking / expertsystemen.

Summary

As a result of the rapid development in computer science, rule based expert systems have entered the application 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 for proving the correctness of the knowledge base. One of these debugging tools, the semantic checker, has been completely revised, resulting in a checker which is capable of *systematically* checking the rule base for *logical completeness and consistency*. This is done by using a method, which is mathematically sound and generally applicable.

Lutgens, J.M.A.
KNOWLEDGE BASE CORRECTNESS CHECKING FOR SIMPLEXYS EXPERT SYSTEMS.
Faculty of Electrical Engineering, Eindhoven University of Technology, The Netherlands, 1990.
EUT Report 90-E-240

Table of contents

1 Introduction

An expert system is a software system which can provide expert problem solving performance in a specific competence domain, by exploiting a knowledge base and a reasoning mechanism.
Expert system technology has developed very fast over the last decade and a huge number of application projects has been started. However, while an impressive and rapidly growing number of expert systems has been produced, the number of real time expert systems is still quite limited. In fact, the development of expert systems largely relies on empirical methods and is not supported by sound and general methologies. This particularly holds for testing the expert system. Expert systems are tested by observing their response in a great number of test cases. This, however, does not guarantee that the expert system is absolutely bug free. A better approach would be to systematically check the expert system.
Expert system debugging is generally concerned with debugging the knowledge of the expert system, located in the knowledge base. Knowledge base debugging is one of the hardest tasks in expert system building. Knowledge base debugging could be facilitated if tools were available for systematically checking the knowledge base. These tools are, however, real hard to build because to debug knowledge you have to understand the *meaning* of the knowledge (semantics). Unfortunately, such tools cannot be built.
What can, however, be built is a checker which understands *logic*. In this paper the development of such a semantic checker, which is capable of understanding logic and deducing logical consequences is described. By using this checker the knowledge base can be checked for *logical* correctness and completeness in a systematic way; a step forward in knowledge base debugging.

2 Expert systems

2.1 Introduction

When computers first emerged from the rapidly advancing electronics industry in the mid-1940s, they were perceived as being nothing more than very large and fast calculating machines. They were ideal for making tedious but simple calculations which otherwise would have demanded the labour of large teams.
As storage capacity expanded, it became clear that the computer was capable of much more. Von Neumann, Turing and many brilliant pioneers had shown that not only could the computer store data, but it could equally well store, retrieve and modify instructions. This opened the door for rapid expansion in engineering design, information analysis and for research of all kinds.

All this led to a new development in computer science : Artificial Intelligence. The main objective of AI is to develop computer programs that are capable of 'human reasoning ' or 'thinking '. According to Minsky, Artificial Intelligence is the science of making machines do things that require intelligence if done by humans.
In the 1960s AI research focused on finding general methods for solving large classes of problems, through attempts to simulate the process of thinking. This approach appeared to be non-fruitful. The more classes of problems a single program could handle, the more poorly it performed on any particular problem. It soon became clear that the power of an AI program was mainly determined by the richness and pertinence of the knowledge it contained, not the way of inferencing. This valuable conclusion was used in AI research and crystallized in what has become to be known as expert systems.

2.2 Expert systems : a general approach

An expert system has been defined as 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].
Thus it is a high performance special purpose system which is designed to capture and use the skill of an expert.

Expert systems typically make use of knowledge relating to the domain in question. This knowledge is acquired from experts and refined in the light of experience. The collection of domain specific knowledge, composed of chunks of knowledge, is called the knowledge base. Often this knowledge is represented as rules and the knowledge base is then referred to as rule base. Rules are a natural formalism for capturing expertise and they have the flexibility required for incremental development.

A rule based expert system consists of three main components :

1] *The knowledge base / rule base*
It contains all the knowledge of the system, in the form of rules.
These rules are often heuristics, rules of thumb.

2] *The database*
It contains specific information about the problem which has to be solved.

3] *The inference engine*
In addition to the rules in the rule base, a mechanism is needed for manipulating these rules in order to solve the problem given the data base. This is done by way of inferencing and the mechanism is referred to as the 'inference engine'.

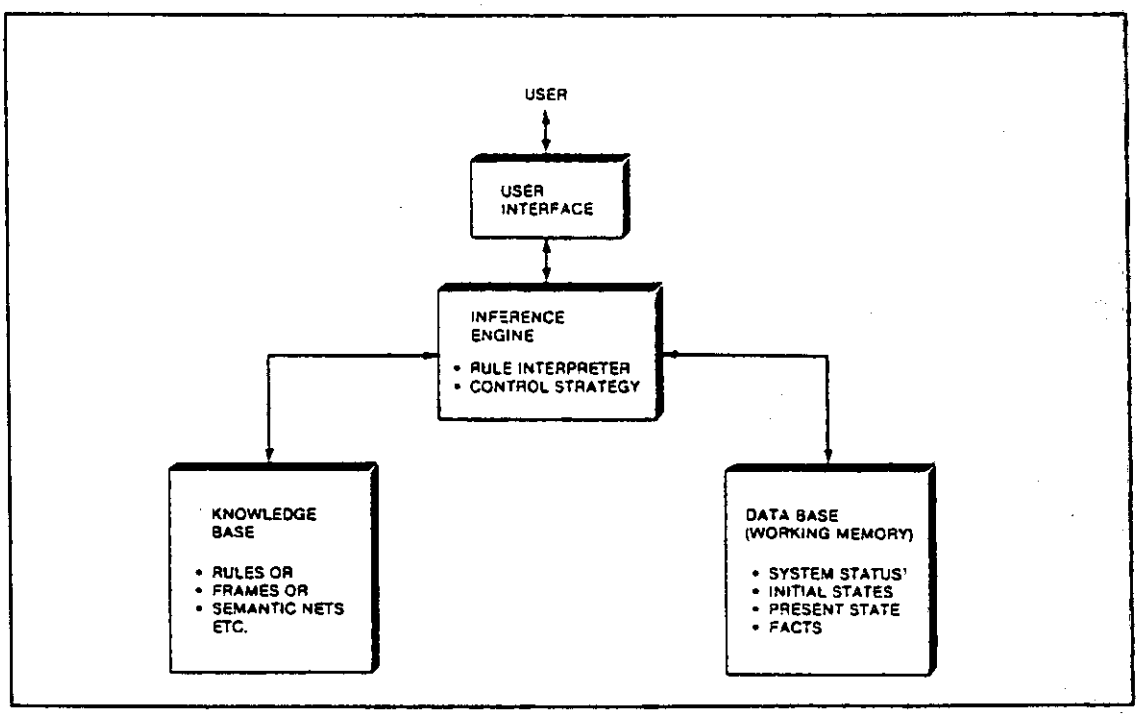


Figure 2.1 Block diagram of a rule based expert system

Rule based expert systems have become familiar tools to solve complex problems. In this field of AI some successful results have already been achieved, for example the medical expert system NEOMYCIN [Hasling, 1984], which can be used by doctors for diagnosing certain infectious diseases and recommending appropriate drug treatment.
Another well known example is DENDRAL [Feigenbaum, 1978], a program for identification of organic compounds by analysis of mass spectrograms.
DENDRAL dates from 1965 but is probably one of the most successful expert systems ever built.

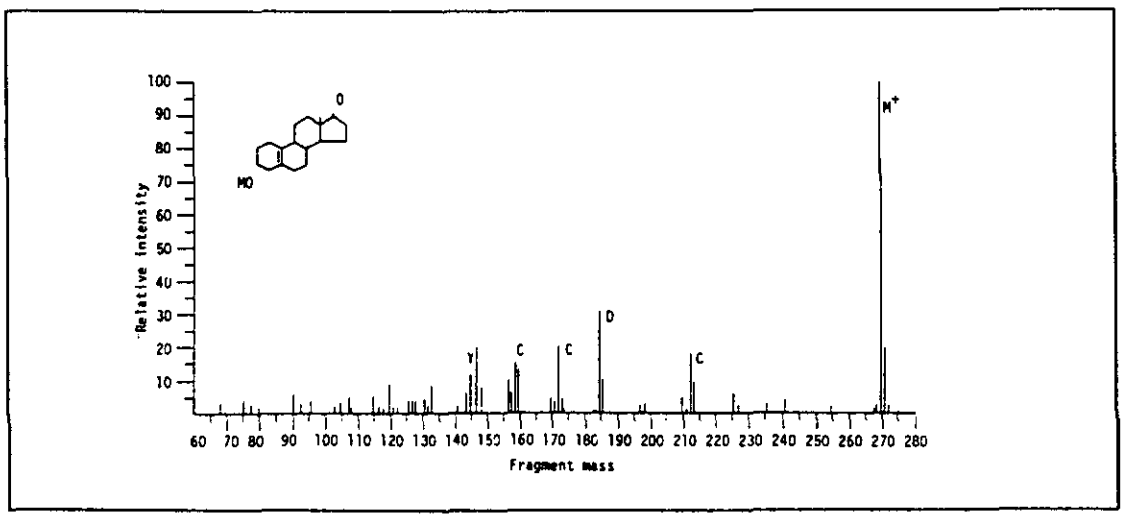


Figure 2.2 An illustration of DENDRAL
Finding the molecular structure using a mass spectrogram
[From Mitchie, 1982]

2.3 Real time expert systems

Expert systems have been constructed for applications to a wide range of problems. For most of these systems, the time aspect is of minor importance. There are however real time applications for expert systems (process control, aerospace, robotics), where the system has to respond before a certain point in time (deadline). An expert system for these real-time problems must, by definition, be able to process incoming data sufficiently rapidly to meet the time constraints. The data must have been processed before new data is supplied. Conventional expert systems, often written in LISP, cannot be used for these purposes because they are too slow (they spend about 85 % of the time solely on searching for the right chunk of knowledge). An expert system that is capable of processing data quickly enough is called a real time expert system.

Real time expert systems are, due to the speed aspect, hard to build.
This explains why the quantity of literature on real time expert systems is not awesome and why just a few have been built.
One of the most promising developments in this field started off as a project five years ago by the division of Medical Electrical Engineering of the Eindhoven University of Technology, under supervision of J.A. Blom, and evolved to what is now called SIMPLEXYS.
SIMPLEXYS is a contraction of Simple Expert Systems. The 'Simple' refers to the ease of using it, not the type of problems it can solve.
SIMPLEXYS is a toolbox for designing real time expert systems, meant for control applications in the medical sector (e.g. patient monitoring). It can, however, also be used for other expert system applications, operating in a dynamical as well as in a static environment.

3 SIMPLEXYS real time expert systems

3.1 History

SIMPLEXYS was not designed, it grew, out of a need [Blom, 1990].
The need arose when research was directed to the design of 'intelligent alarms' in anaesthesia. Conventional alarms are clumsy at best, sometimes even useless. Performance had to be improved and this could be done by using an expert system. This expert system should, however, have some special features :

1] Capable of operating in real time.

2] Easy to use and understand, even for non-programmers.

3] Compact enough to run on a PC.

4] Efficient, fast and above all reliable.

5] Offer sufficient potentials to check correctness.

Unfortunately, such a system was not available. Thus a new expert system had to be developed, to fulfil the requirements above.
A special toolbox was created in order to facilitate development. This toolbox evolved to what is now called SIMPLEXYS : a toolbox for designing real time expert systems. SIMPLEXYS is written in Pascal, although a C version is also available. In contrast to LISP, Pascal is a very efficient programming language, producing fast executable programs in which no time is wasted on searching, thus making real time applications feasible.

3.2 The SIMPLEXYS programming language

In order to better understand the remainder of this text, the SIMPLEXYS programming language is introduced here. The material presented in this section is intended to serve as a review. For a thorough treatment of the subject, see [Blom, 1990]. These references can also be used to become more familiar with SIMPLEXYS in general.

SIMPLEXYS is based on three-valued logic. A rule can either have the value TR (true), PO (possible or unknown), or FA (false).
How rules are constructed in SIMPLEXYS is expressed by the following example:

```
ADULT : 'The person is an adult'
ASK
THEN FA : CHILD
```

The first line denotes the name of the rule (ADULT). Symbolic names are used instead of numbers to improve readability.
The second line states the rule type; in this case it is an ask rule.
The third line is optional, but can be used as extra information, which becomes available as soon as the rule is assigned a value. In this case it states that if ADULT is true, then CHILD has to be false.

3.2.1 Rule types and value assignment

The collection of rules (rule base) can be converted into a semantic network. A set of junctions (the rules) are mutually connected. The connection represents the relationship between rules. There are two kinds of rules in SIMPLEXYS :

1] *Primitive rules*
These rules are independent of other rules and get their value by some sort of direct assignment.

2] *Evaluation rules*
These rules operate on a higher level and are dependent of other rules (either primitive or other evaluation rules).
This can best be illustrated by an example :

TIGER : 'The animal is a tiger'
MAMMAL and CARNIVORE and TAWNY and BLACKSTRIPED

If all of the 4 properties (mammal, carnivore, tawny, blackstriped) have the value TR (true), then the animal is considered a tiger.

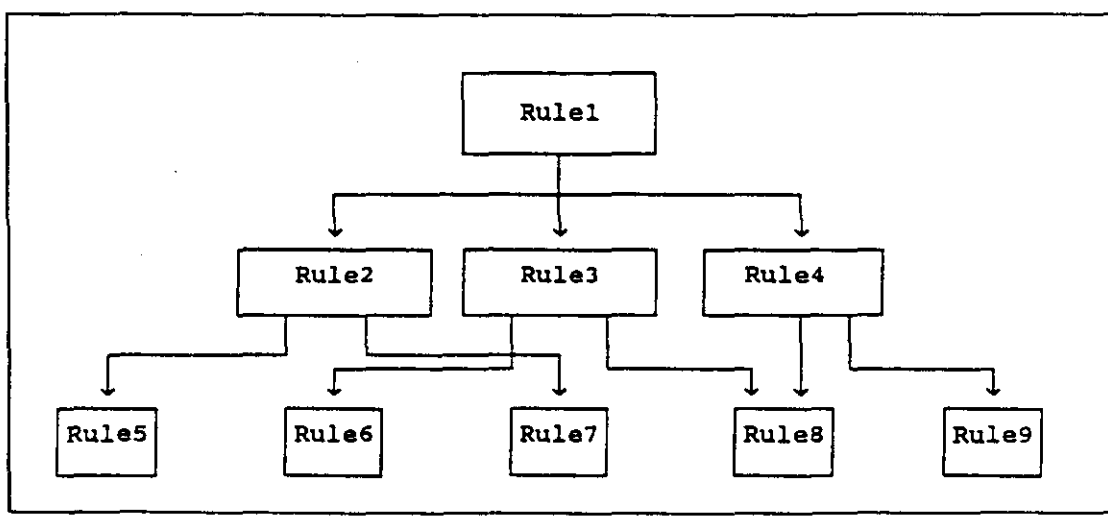


Figure 3.1 Semantic network of rules
Rules 1 to 4 are evaluation rules, rules 5 to 9 are primitive rules

Rules are evaluated only once in SIMPLEXYS. This is done in a recursive way. In order to determine the value of an evaluation rule, the values of the

constituent rules have to be obtained first. When these are evaluation rules themselves, evaluation is again in order. This recursive process ends when the primitive rules are reached, which get their value by direct assignment (i.e. the result of a test, the answer to a question). This type of evaluation is called *backward chaining*.

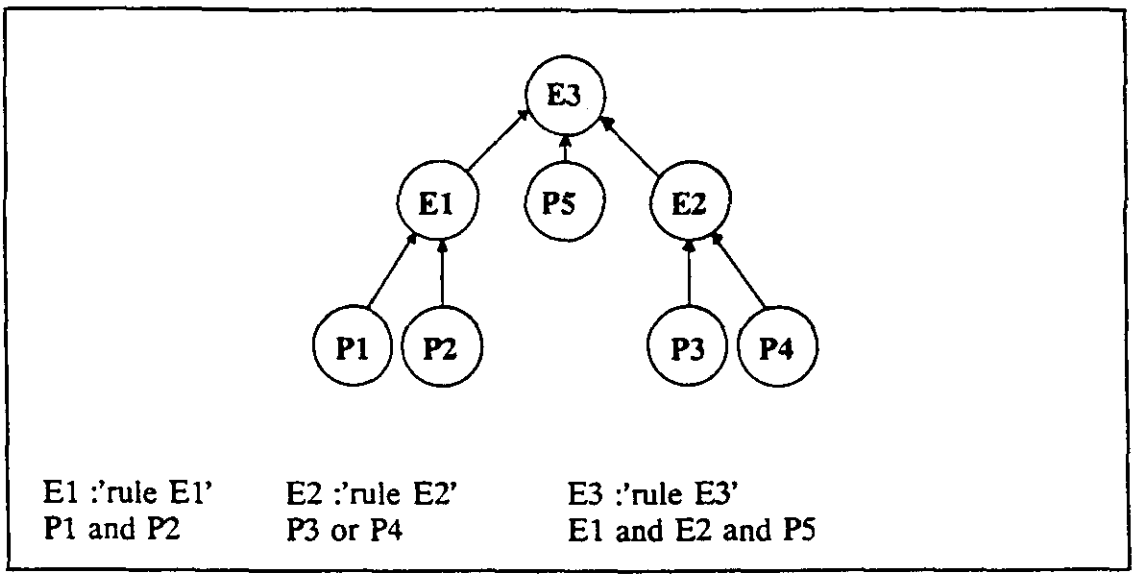


Figure 3.2 Evaluation of E3 using backward chaining
E denotes evaluation rule, P primitive rule

A rule can also be given a value by forward chaining :

CAR :' The object is a car'
ASK
THEN GOAL : VOLKSWAGEN

The combination *then goal* states that if CAR becomes true, rule VOLKSWAGEN should be evaluated.
The collection of then/else/ifpo (if possible) is called *thelses*, and is used in combination with TR/PO/FA or GOAL (e.g. else tr, then goal, ifpo then fa).
Powerful rule bases can be built by using forward- as well as backward chaining.

Primitive rules are assigned a value in a way that depends on the type of the rule. In SIMPLEXYS five different primitive rules types are used :

1] Fact rules (FACT), which have a constant and unchanging value (TR,PO,FA).

2] Ask rules (ASK), which are given a value by asking the user (TR,PO,FA). The answer is entered by using the keyboard.

3] Test rules (TEST), used for testing data through Pascal interfacing. The result of the test is either TR, PO or FA. Test rules are useful in control applications. A special test rule is the binary test (BTEST), which can only result in TR or FA.

4] Memo rules, used as memory, can only be given a value by other rules e.g. through a then tr. Memo rules can either be TR, PO or FA.

5] State rules, denoting a context, assigned a value initially or via the protocol. State rules are either TR or FA, but never PO. For more details see [Lammers,1990].

In the next section the logic used in SIMPLEXYS will be discussed.

3.2.2 SIMPLEXYS logic

The logic used in SIMPLEXYS is very much like boolean logic. Boolean logic is easy to use and fast, which makes it suitable for real time applications.
Opposed to boolean logic, SIMPLEXYS uses three-valued logic instead of two-valued logic (TR/FA). The use of three valued logic can be justified by the fact that it agrees better with human reasoning, for not everything is TR or FA in the real world. Sometimes we just don't know and that is where the third value PO fits in. Possible is used whenever a rule is neither provably true or provably false.

Expressions in SIMPLEXYS consist of two entities, propositions (also called variables) and operators. The operators are either monadic (one argument) or dyadic (two arguments).

3.2.2.1 Monadic operators

Simplexys has the following monadic operators :

NOT R : The negation of R
MUST R : R is guaranteed to be true
POSS R : No definite value can be found for R

in which R represents a rule of any type.

Table 3.1 Truth table monadic operators

x	not x	must x	poss x
TR	FA	TR	FA
FA	TR	FA	FA
PO	PO	FA	TR

3.2.2.2 Dyadic operators

SIMPLEXYS supports the following dyadic operators :

AND, UCAND, OR, UCOR, ALT

AND and OR are already known from boolean algebra. The difference between AND and UCAND is that the latter is evaluated unconditionally. In x AND y, y is not evaluated if x equals FA; by using x UCAND y both will be evaluated. The same holds for OR and UCOR if x equals TR.
The ALT operand is not known in boolean algebra. ALT stands for 'logically equivalent alternative'. In the expression x ALT y, y is used as an alternative for x whenever the value of x cannot be determined (x has the value PO). The arguments of an ALT operator may never take on opposite values, for they have to be logically equivalent.

Table 3.2 Truth table dyadic operators

x \ y	TR	FA	PO
TR	TR	FA	PO
FA	FA	FA	FA
PO	PO	FA	PO

and/ucand

x \ y	TR	FA	PO
TR	TR	TR	TR
FA	TR	FA	PO
PO	TR	PO	PO

or/ucor

x \ y	TR	FA	PO
TR	TR	**	TR
FA	**	FA	FA
PO	TR	FA	PO

alt (** = conflict)

3.2.2.3 The History operator

Each rule has a history counter, which contains the period, in seconds, during which its value has remained unchanged. Thus we are able to answer questions like 'how long has rule x been true ?'.
History operators are used as follows : *rule history-op* (numerical expression)
For example, the value of the expression

HIGHBLOODPRESSURE > (120)

is true if the blood pressure has been high for more then 120 seconds.
Six history operators can be used in SIMPLEXYS :

=	**equal**	< >	**not equal**
>	**greater than**	> =	**greater than or equal**
<	**less than**	< =	**less than or equal**

One application of history operators is, that rules can be used to detect stable situations.

3.2.2.4 Priority of operators

In SIMPLEXYS, history operators have the highest priority, followed by the monadic operators (which all have the same priority) which again have higher priority than the dyadic operators (which all have the same priority too). Priority however can be forced in SIMPLEXYS by using parentheses :

(X and Y) or (not Z and (U alt V))

Parentheses should be used whenever one is not sure about the correct priority, to prevent an incorrect evaluation.

Note that in boolean logic the priority of the AND is usually taken to be higher then the OR. This is not the case in SIMPLEXYS.

3.2.3 The SIMPLEXYS rule base

In the previous section the SIMPLEXYS programming language syntax was discussed. Rules in the rule base have to be expressed according to this syntax.

The rule base however is composed of more than just rules; in fact the rule base consists of up to 7 sections, each indicated by a special keyword :

1] DECLS : Declarations
2] INITG : Global initializations
3] INITR : Run initializations
4] EXITR : Run exit code
5] EXITG : Global exit code
6] RULES : Rule definitions
7] PROCESS : Protocol, describing the dynamics of the system.

The first 5 sections are optional, section 6 and 7 are mandatory.
Use and function of each section will only be discussed if relevant for this document.

3.3 The SIMPLEXYS Toolbox

Once the rule base has been created, it has to be linked with the inference engine to obtain a working expert system. Conversion from rule base to expert system is done by the SIMPLEXYS Toolbox. This toolbox does three things :

1] It converts the rule base into an appropriate form for the inference engine, by using a rule compiler.

2] It checks the rule base for correctness and completeness.
This is an very important aspect, for errors in the rule base will lead to erroneous behaviour of the expect system which of course is unacceptable.

3] It builds the expert system by linking the (compiled) rule base with the inference engine.

These three aspects are incorporated in the following six tools which together form the SIMPLEXYS toolbox :

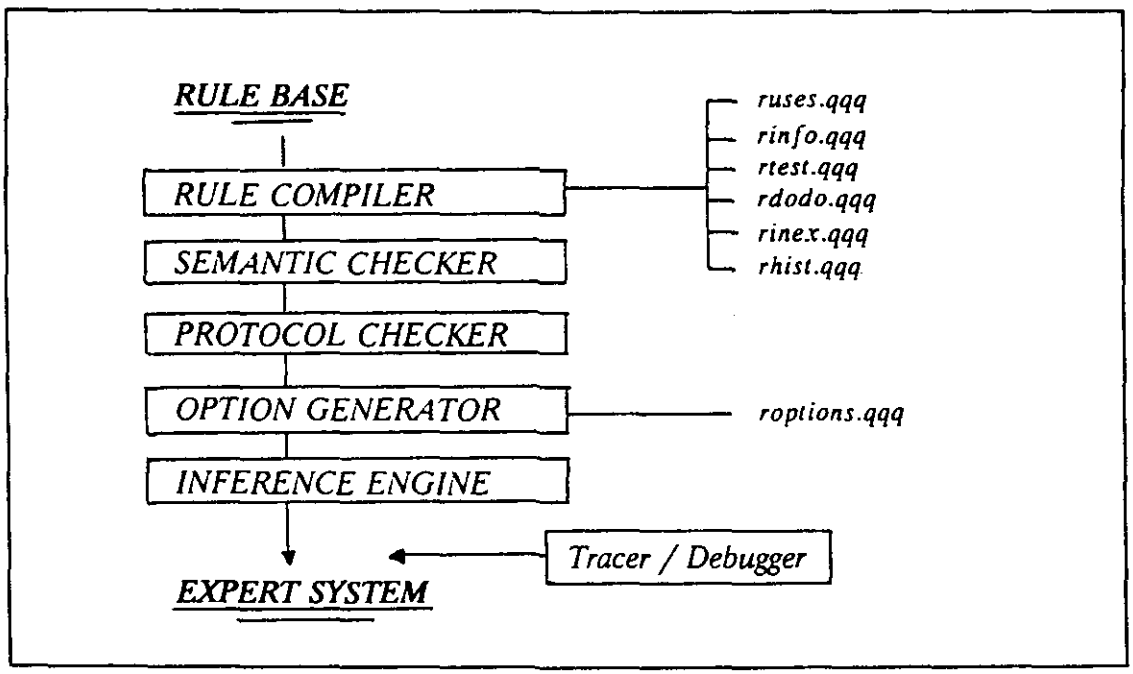


Figure 3.3 The SIMPLEXYS toolbox

1] *The rule compiler*
The rule compiler translates the rule base into an internal representation of six so called qqq-files :

1) rinfo.qqq : Contains all the arrays and tables used for representing the rules and their mutual connectivity.

2) rtest.qqq : Contains all the test sections defined in the test rules.

3) rhist.qqq : Contains the information about the history sections.

4) rdodo.qqq : Contains the collection of DO sections used in the rule base.

5) rinex.qqq : Contains the initialization sections and exit sections.

6) ruses.qqq : Contains the Turbo Pascal units used by the rule base.

The compiler also checks for the following syntax errors and some very simple semantic errors :

- mistakes in rule constructions.
- duplicate rule names.
- internal overflow.
- unconnected rules.
- incomplete rule set.
- no state rule initially true.

2] *The semantic checker*
The semantic checker performs several semantic checks and generates appropriate messages if errors are detected. Semantic checking is a powerful tool for (partially) proving rule base correctness.

3] *The protocol checker*
The protocol checker is used for detecting errors in the process description part or the protocol. Protocol checking is done quite extensively and covers syntax, topology, as well as dynamic errors. For more details see [Lammers, 1990].

4] *The option generator*
With the option generator several run time options can be selected for the inference engine.

5] *The inference engine*
The SIMPLEXYS inference engine actually builds the expert system by combining the output of the rule compiler with inference processes into one program. In this phase the Pascal compiler checks the Pascal sections for correctness. The expert system is now ready to run.

6] *The tracer/debugger*
The tracer/debugger is a tool to examine the inferencing process of the expert system while it processes symbolic information. The tracer/debugger can be used *after* the expert system has been built.
For more details see [Philippens, 1990].

A rule base that passes step 1 to 5 can be converted into a working expert system. This however does not necessarily mean that the expert system will function in a correct fashion. There might still be bugs in the system, undetectable (yet) for the SIMPLEXYS toolbox.

Finding these bugs and fixing them is called debugging and will be discussed in the following chapter.

4 Debugging expert systems

4.1 Introduction to expert system debugging

Expert systems are used for various applications. Some of these applications even involve critical decision making, where the user totally relies upon the judgement and competence of the system. Therefore expert systems have to be absolutely fail-safe; they may never give an incorrect solution or recommend an erroneous course of action.

Unfortunately expert systems are developed by humans, and humans make mistakes, which implies that expert systems can make mistakes too. Checking for erroneous behaviour of the system and fixing it is called debugging and is one of the hardest tasks in expert system development.

One way of debugging an expert system is by observing the system's behaviour in a great number of test cases. Although this is an essential part of testing the expert system, it will never guarantee that all bugs will be detected (unless of course the amount of test cases is exhaustive).
Reliability can be improved by using debugging tools (programs especially designed for detecting certain kind of bugs), which check the expert system systematically.
Debugging tools are much more powerful than test cases; systematically checking enables them to detect *all* the bugs (of the kind in question) in the system.
Unfortunately debugging tools can only be realised for certain kinds of bugs.
Debugging should therefore incorporate both, debugging tools as well as test cases, to achieve maximal reliability.

Debugging rule based expert systems can be divided into three parts :

1] *Rule base debugging*

2] *Data base debugging*

3] *Inference engine debugging*

Each part will be discussed; the emphasis, however, is on rule base debugging.

4.1.1 Rule base debugging

The rule base represents the knowledge of the expert system; therefore rule base debugging is equivalent to knowledge base debugging. Knowledge base debugging can be better understood if we know how the knowledge is acquired.
Knowledge for an expert system is acquired from one or more experts in the domain. Gathering and organizing the knowledge into a appropriate form is the job of the *knowledge engineer [Frenzel, 1987]*.

A knowledge engineer is a unique individual. That person is not an expert in the domain of the expert system, but is capable of understanding the domain and learning the problem solving process within it. A knowledge engineer knows and understands expert systems and can take knowledge in its various forms and convert it into rules appropriate for the expert system. Knowledge engineers are scarce, but they are absolutely essential to the creation of an expert system. This is particulary true for medium to large expert systems.
The first task of a knowledge engineer is to find a person who is an expert in the domain. Next thing to do is extracting the knowledge from the expert. This is done by interviewing. Obviously in most cases one interview is not enough. In fact, the knowledge acquisition process generally will take many weeks or even months. The knowledge engineer will conduct a series of interviews with the expert, while defining the scope of knowledge, identifying the kind of problems being solved and determining the knowledge and approaches required to solve them. These sessions will be of the question - answer type.
Another approach is to present the expert with one or more problems to solve. Given the problem, what does the expert *think* and *do* to solve it? The knowledge engineer must try to get at the thought process to determine what information is required to solve the problem and how that information is related to the knowledge the expert has.
There is usually a specific sequence of information processing and problem solving. Important to remember in interviewing an expert is that most experts do not truly fully understand how they go about solving their problems. In fact it has been said that the better the experts, the less they know about their actual problem solving technique. Superior experts have years of experience and in-depth knowledge about the domain and the problem solving process is a natural, integrated thing that is difficult to uncover. The danger in interviewing experts is that you will force them to think of the problem solving process they use. They probably do not understand it themselves but they will attempt to give some approach or sequence. The knowledge engineer will accept this but must examine it sceptically.

Once the knowledge engineer has gathered enough information, he can start organizing it, make sense out of it, and ultimately convert it into rules.
Conversion of knowledge into rules is a complex and time consuming task. There is no general approach to use, although some guidelines can be given. Subdividing the knowledge into smaller parts, so-called chunks of knowledge, can for instance make things easier for the knowledge engineer. Step-wise development of the knowledge base is another good idea; catching errors is always easier in the early development stages.
Five types of problems explain most of the errors in rule base construction :

1] The expert neglected to express rules to cover all the special cases that arise.

2] The knowledge engineer's interpretation of the expert's knowledge is erroneous.

3] The knowledge engineer overlooked some of the expert's advice.

4] Some of the knowledge could not be expressed in the form of rules.

5] Syntax errors like misspelled words, illegal rule construction or handwriting slips.

Keeping this in mind while constructing a rule base will probably result in fewer errors.

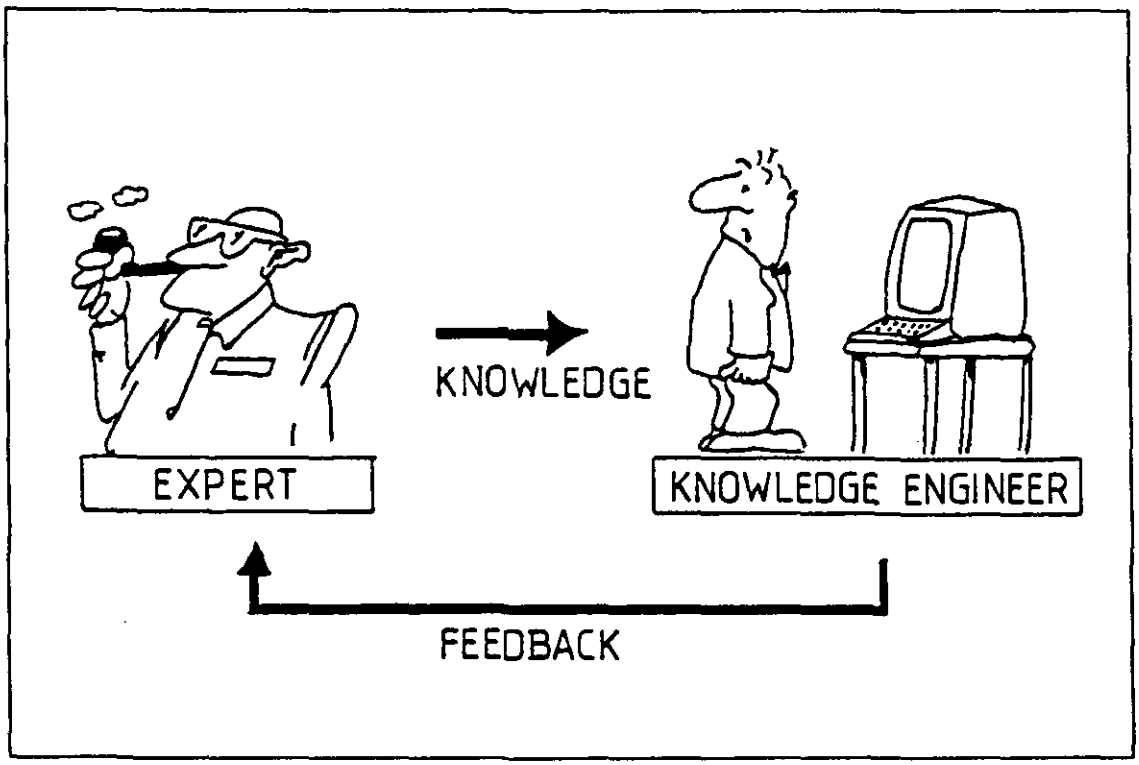


Figure 4.1 Knowledge acquisition

The next step to follow is debugging the rule base. This process involves testing and refining the rule base in order to discover and correct a variety of errors that can arise during the knowledge acquisition phase.
Regardless of how an expert system is developed, its developers can profit from a *systematic* check on the rule base. This can be accomplished by a program that checks the rule base for *completeness and consistency* during the system's development.

Checking for completeness
Incompleteness of the knowledge is the result of missing rules. If rules are missing then a situation exists in which a particular inference is required, but there is no rule that succeeds in that situation and produces the desired conclusion. In other words : Does the rule base contain all the rules needed to produce the desired conclusions?

The program that must check the rule base has, in general, no knowledge of the problem domain and is not capable of checking the semantics in the rule base. Therefore such a program cannot guarantee a completely correct rule base; it can only improve our confidence in a correct behaviour.

Checking for consistency
When knowledge is represented in rules, inconsistencies in the rule base appear as follows :

1] *Conflicts* : A conflict occurs when two or more rules succeeding in the same situation give conflicting results. This situation can lead to inconsistent or even erroneous behaviour of the expert systems.

2] *Redundancies* : Redundancy occurs when two or more rules succeed in the same situation and give the same results. Although this case normally does not cause erroneous behaviour, it points out that the knowledge base can be simplified.

3] *Subsumption* : Subsumption occurs when two or more rules have the same results, but one contains additional restrictions on the situations in which it will succeed. In some situations this will lead to redundancy.

Of the three types of inconsistency described above, only conflicts are guaranteed to be true errors. In practice, redundancy and subsumption may not cause problems. Nevertheless, it may be interesting to find redundant rules or subsumptions, because they usually indicate an implementation error, or otherwise they points out that the rule base can be simplified.

Once the rule base has been debugged and put into its final form, it can be used for building an expert system. However, its development does not end here. Most domains are dynamic and new knowledge must be added constantly. This means that the rule base has to be modified. Usually new rules will be added,while old rules will be deleted or modified. Maintenance and updating are however dangerous, because new rules may lead to conflicting situations in a rule base. This is often the case if maintenance and updating is done by several people. A rule base which has been modified should therefore be tested again for completeness and inconsistency to ensure correctness.

4.1.2. Database debugging

Another important part of an expert system is the data base. The data base contains all specific information about the problem to solve. Data base debugging however is not concerned with the expert system itself, but with errors in the problem definition. A problem that is not well defined, will lead to an incorrect solution. Therefore special attention should be paid to formulating the problem and converting it into an appropriate form for the expert system. The same holds when consulting a real expert; if you cannot define your problem properly, he will most likely give an incorrect answer.

Most of the problems in expert systems, though, are caused by an incomplete or incorrect problem definition. This is, however, a user's problem, not an expert system's problem, and therefore it should be solved by the user.

4.1.3. Inference engine debugging

The inference engine uses the knowledge stored in the rule base to solve the problem given in the data base. Now assume that the rule base is correct and complete and the problem in the data base is well defined. Then the inference engine should come up with the right solution. I say 'should' because the inference engine may contain some bugs, leading to incorrect inferences and thus inaccurate solutions. Bugs in the inference engine however can easily be traced because the inference processes can be described mathematically.

Regarding the problem of debugging expert systems, we may conclude that rule base debugging is the most important one and also the most difficult one.
Bugs in the inference engine can easily be detected by using common sense and applying logic, while bugs in the rule base are harder to trace.

In the next section we will discus the debugging tools designed for SIMPLEXYS.

4.2 SIMPLEXYS expert systems debugging tools

The SIMPLEXYS toolbox is equipped with several debugging tools :

4.2.1 The rule compiler

Apart from compiling the rule base the rule compiler also performs the following checks on the rule base :

* There are no rules.
* There are no STATE rules.
* illegal rule syntax.
* duplicate rule names.
* no STATE rule is INITIALLY TR.
* There are no ON statements.
* A FROM or TO list contains a non STATE rule.
* A rule is unconnected (in no way used by other rules).
* Incomplete rule set; a rule is needed for evaluation but has not been defined.

These checks are, although very simple, quite useful. Many of the common errors in the rule base are detected by the rule compiler.

4.2.2 The semantic checker

The SIMPLEXYS semantic checker performs several semantic checks on the rule base by using the information stored in rinfo.qqq. The six semantic errors checked by this program are :

1] *Self referencing evaluation loops :*

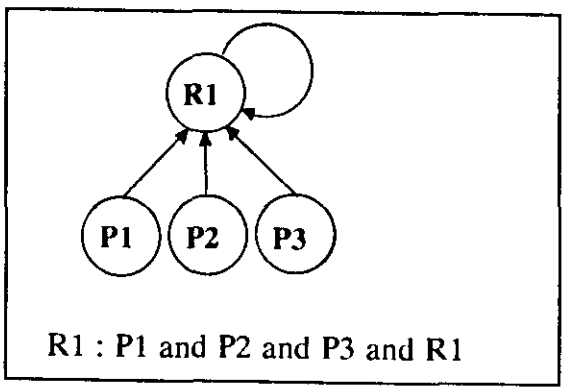


Figure 4.2 A self-referencing evaluation loop

Whenever a rule is part of its own evaluation expression, evaluation is never-ending. Loops will seldomly occur in a rule base, but they can be hidden by the extent of the rule base.

2] *Thelses loops :*

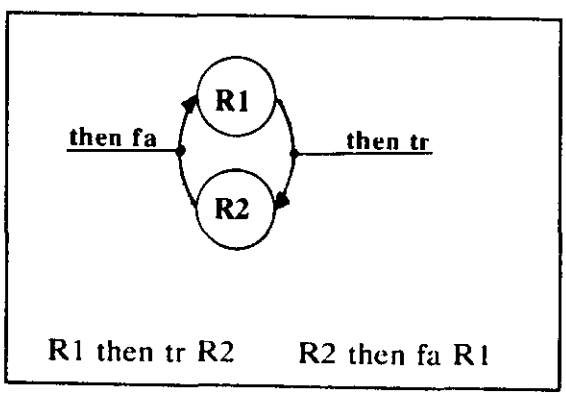


Figure 4.3 A thelse loop

Thelses can also form loops. In this example a conflict occurs : If rule R1 evaluates to true then rule R2 becomes true, thus setting R1 to false (opposite assignment).

3] *Conflicting thelses :*

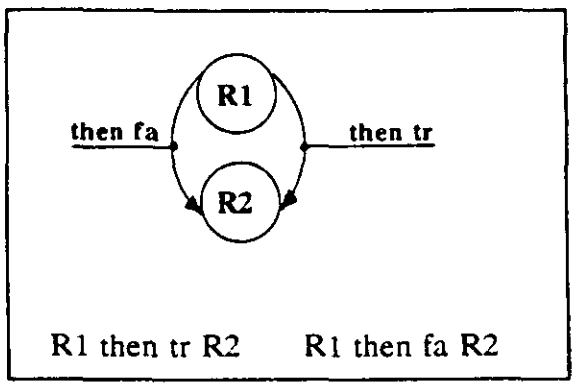


Figure 4.4 Conflicting thelses

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

4] *Thelses to successor :*

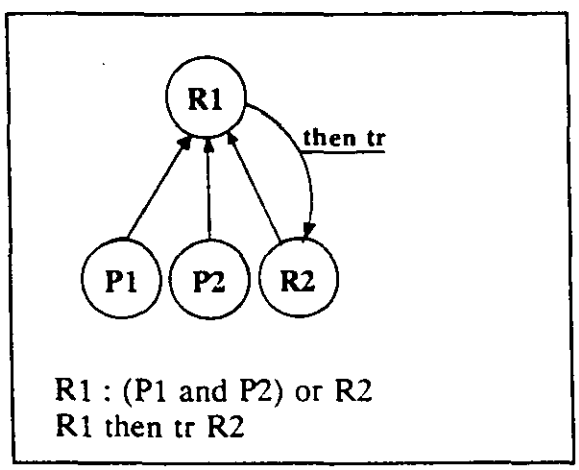


Figure 4.5 Thelses to successors

Rule R1 uses rule R2 for its evaluation and does a then tr to it. This may result in a conflict (i.e. if P1=tr, P2=tr, R2=fa).

Note : There will be no conflict if the rule is a memo , because an assignment to a memo rule is postponed to the start of the next run. State rules are syntactically forbidden in such constructs.

5] *Thelses to predecessors :*

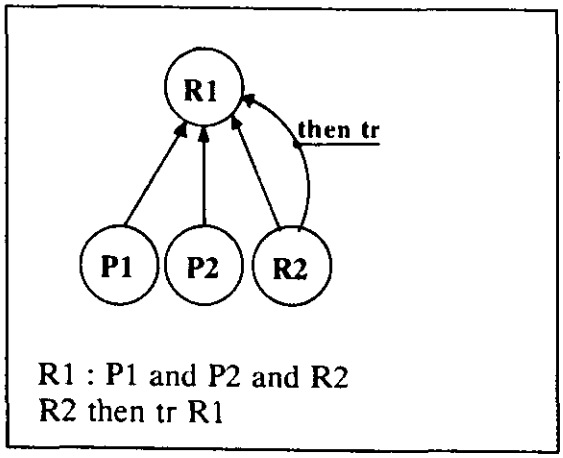


Figure 4.6 Thelses to predecessors

Rule R1 uses rule R2 for its evaluation and R2 does a then tr to R1. This may result in a conflict (i.e. if P1=fa, P2=fa, R2=tr).

6] *Unconnected non-STATE rules :*

A warning will be generated if a certain rule is in no way used in the process :

* The rule is not used in any evaluation.
* The rule is not thelsed by any rule.
* The rule is not a trigger rule.

4.2.3 The protocol checker

The protocol checker is designed for detecting errors in the process description part or the protocol of the rule base. It checks the protocol on three different types of errors :

1] Syntax errors :
- No start states; no rules are initially tr.
- No end states; no ON statement has an empty TO list.
- Conflicts at states; two transitions have equal FROM lists and the same trigger.
- Empty prestate; each state must be in at least one TO list.
- Empty poststate; each state must be at least one FROM list.

2] Topology errors :
- Self loops; the FROM list and the TO list of an ON statement are not disjunct.
- Identical ON statements; two ON statements have the same FROM list and the same TO list.

- Identical states.
- Net part not connected to start state.
- Net part not connected to end state.

3] Dynamic errors :
- Deadlock; there is a firing sequence resulting in a context where no further change of state is possible.
- 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.
- System cannot stop; there is no firing sequence making only end states true.
- Not all transitions can fire at least once.
- Conflicts; conflicts arise when the FROM lists have a non empty intersection.

Protocol checking is quite extensive and one has to be familiar with Petri-nets to understand it. This will not be discussed here; for more details see [Lammers,1990].

The three knowledge base debugging-tools discussed here (rule compiler, semantic checker, protocol checker), have been proven to be very useful. The semantic checks, however, are very limited. The semantic checker can, however, become the most valuable tool in debugging, if properly expanded. In the following chapter such a semantic checker, with a much better performance, will be discussed.

5 Designing a semantic checker

5.1 Introduction

As we saw in the previous chapter, many errors can arise in the process of transferring expertise from a human expert to a rule base. It is therefore very important to check the correctness of the rule base. Unfortunately no formal mathematical methods are available to analyze the deep knowledge of the rule base. Nevertheless we would like to have a tool to check the rule base for consistency and completeness. Not many of these tools are available, and when they are, their quality is not always adequate.
The same holds for the semantic checker. It can handle only a few conflict situations and the overall performance could be improved.
How well a semantic checker can perform depends upon the structure of the rules themselves and the rule base as a whole. What we would like to achieve is that the semantic checker checks for consistency by not only checking for conflicts, but also for redundancy and subsumption. The SIMPLEXYS language used in the rule base offers enough potential to make this feasible. A true semantic checker can however never be built, for 'semantics' refers to the *meaning* of the rule and a checker will never be able to '*understand*' the rules.
What we *can* do, is to make the semantic checker '*logically understand*', thus restricting correctness checking to checking for *logical completeness and consistency* of the rule base.
In practice, however, checking for logical correctness is almost equal to checking for correctness because the rule base is logically constructed.
Logical correctness checking can be done if the semantic checker understands logic and is able to make logical deductions.

Propositional logic describes the relationship between logic and the way how rules are represented and some insight in to proportional logic could be useful for designing a semantic checker that understands logic. Therefore this topic will be discussed in the next section.

5.2 Propositional logic

Logic, which was one of the first representations schemes used in AI, has two important and interlocking branches. The first is consideration of *what can be said*, what relations and implications one can formalize. These are called the axioms of the system. The second is the *deductive structure*, the rules of inference that determine what can be inferred if certain axioms are taken to be true.
Logic is quite literally a formal endeavour; it is concerned with the form, or *syntax*, of statements and with the determination of truth by syntactic manipulation of formulas. The expressive power of a logic-based representational system results from incremental development. One starts with a simple notion (like that of truth or falsehood) and, by inclusion of additional notions (like conjunctions and predication) develops a more expressive logic; one in which more subtle ideas can

be represented. The SIMPLEXYS rule base is also built in this manner.
The most fundamental notion in logic is that of truth. A properly formed statement or *proposition* has one of two different possible *truth values* TRUE or FALSE (Note : In SIMPLEXYS another value is allowed : PO , but the influence of this will be discussed later).
Typical propositions are *Bob's car is blue, John is Mary's uncle* or *the patient is not breathing*. Note that each of the sentences is a proposition, not to be broken down into its constituent parts. Thus, we could assign the truth value TRUE to the proposition *John is Mary's uncle* with no regard for the *meaning* of *John is Mary's uncle*, that is that John is the brother of one of Mary's parents.
Propositions are those things that we can call TRUE or FALSE. Terms such as *Bob's car, breathing* would not be propositionals, as we cannot assign a truth value to them. Pure disjoint propositions are not very interesting. Many of the things we say and think about can be represented in propositions that use *sentential connectives* to combine simple propositions.
There are five commonly employed connectives :

AND	$\wedge$
OR	$\vee$
NOT	$\neg$
IMPLIES	$\rightarrow$
EQUIVALENT	$\blacksquare$

The use of the sentential connectives in the syntax of propositions brings us to the simplest logic, *propositional calculus*, in which we can represent statements like *the patient needs respiration* or *if the bloodpressure is to high then increase SNP flow*. If X and Y are any two propositions then :

$X \wedge Y$ *is : TRUE if X is TRUE and Y is TRUE; otherwise X ∧ Y is FALSE .*

$X \vee Y$ *is : TRUE if either X is TRUE or Y is TRUE or both; otherwise FALSE.*

$\neg X$ *is : TRUE if X is FALSE, and FALSE if X is TRUE.*

$X \rightarrow Y$ *is : TRUE if Y is TRUE or X is FALSE; otherwise FALSE.*

$X \blacksquare Y$ *is : TRUE if both X and Y are TRUE, or both X and Y are FALSE; otherwise FALSE.*

The following table summarizes these definitions.

Table 5.1 Truth table of some connectives

X	Y	X ∧ Y	X ∨ Y	X → Y	X ≡ Y	¬ X
T	T	T	T	T	T	F
T	F	F	T	F	F	F
F	T	F	T	T	F	T
F	F	F	F	T	T	T

From syntactic combination of variables and connectives, we can build sentences of proportional logic, just like the expressions in human language. Some typical sentences are :

(1) $X \rightarrow (Y \wedge Z)) \equiv ((X \rightarrow Y) \wedge (X \rightarrow Z))$

(2) $\neg (X \vee Y) \equiv \neg (\neg X \wedge \neg Y)$

(3) $(X \wedge Y) \vee (\neg Y \wedge Z)$

Sentence 1 is a *tautology*; it states , 'Saying X implies Y and Z is the same as saying that X implies Y and X implies Z'. *Tautologies* are very special because they are always true no matter what propositions are substituted for X, Y and Z. This can be compared to the sentence 'Tomorrow it will rain or it will not rain', logically represented by (X ∨ ¬ X).
We do not have to wait for tomorrow to say that the sentence is true; we are able to say that it is true beforehand. This is the strength of tautologies; the actual values of the propositions do not have to be known to assign a truth value to the sentence.

Sentence 2 is a *contradiction*. No matter what assignment of values is used, the sentence is always false.

Sentence 3 is neither a tautology nor a contradiction, we have to know the values of X, Y and Z to determine its truth value.

Using tautologies and contradictions seems to be absurd but they are very useful for checking the rule base, as we will see later on.
In propositional calculus, we also encounter the first *rules of inference*. An inference rule allows the deduction of a new sentence from previously given sentences. The power of logic lies in the fact that the new sentence is assured to be true if the original sentences were true. The best known inference rule is *modus ponens*. It states that if we know that X is TRUE and we know X→Y then we know that Y is TRUE. For example, if we know that the sentence *John is an uncle* is true and we also know that *all uncles are male* then we can conclude that *John is a male.*

The SIMPLEXYS language used in the rule base is quite similar to propositional calculus. We can therefore apply theorems of the propositional calculus to the rule base.

Table 5.2 Theorems in propositional logic

(1)	$X \rightarrow Y$	=	$\neg X \vee Y$
(2)	$X \equiv Y$	=	$(X \rightarrow Y) \wedge (Y \rightarrow X)$
(3)	$X \equiv Y$	=	$(\neg X \vee Y) \wedge (\neg Y \vee X)$
(4)	$\neg \neg X$	=	X
(5)	$X \wedge Y$	=	$Y \wedge X$
(6)	$X \vee Y$	=	$Y \vee X$
(7)	$\neg (X \wedge Y)$	=	$\neg X \vee \neg Y$
(8)	$\neg (X \vee Y)$	=	$\neg X \wedge \neg Y$

ad 1 : This can be verified by examining the truth table

Table 5.3 Truth table of ($X \rightarrow Y$) and ($\neg X \vee Y$)

X	Y	(X → Y)	(¬ X ∧ Y)
T	T	T	T
T	F	F	F
F	T	T	T
F	F	T	T

ad 2 : X is equivalent to Y if X implies Y and Y implies X. This can be verified by examining the truth table.

ad 3 : This directly follows from (1) and (2).

ad 5,6 : The commutative law.

ad 7,8 : De Morgan's laws.

We will conclude this paragraph with some important definitions and theorems, which will later be used for proving consistency of a knowledge base.

Definition : A sentence is said to be a *tautology* if and only if it is true under all interpretations.

Definition : A sentence is said to be a *contradiction* if and only if it is false under all interpretations.

Theorem 1 : If two sentences, X and Y, are logically equivalent then (X ≡ Y) is a tautology. (Proof by definition).

Theorem 2 : If a sentence X implies another sentence Y then (X → Y) is a tautology. (Proof by definition).

Theorem 3 : If a sentence, containing only the connectives ∧,∨,¬, is a tautology or contradiction then at least one of the propositions in the sentence has to occur in a positive sense as well as a negative sense.
A term occurs in a negative sense when it is part of a not expression e.g. *not (P1 and P2)*, where both P1 and P2 occur in a negative sense.

Proof
Lets assume that each proposition (P, Q, R, S, V,....) in the sentence E occurs in a positive sense or in a negative sense, but not both, then theorem 3 states that E cannot be a tautology or contradiction.

Proof a : E cannot be a tautology
The sentence E can be written as E (P, Q, ¬ R, S, ¬ V, ... ∧, ∨).
Now without loss of generality we may assign the value false to the propositions occurring in a positive sense and the value true to the propositions occurring in a negative sense. We then get E (fa, fa, ¬tr, fa, ¬tr,... ∧, ∨).
By now substituting fa for ¬tr we get E (fa, fa, fa, fa, fa,... ∧, ∨) which represents a sentence with the value false; thus E cannot be a tautology.

Proof b : E cannot be a contradiction
The sentence E can be written as E (P, Q, ¬ R, S, ¬ V, ... ∧, ∨).
Now without loss of generality we may assign the value true to the propositions occurring in a positive sense and the value false to the propositions occurring in a negative sense. We then get E (tr, tr, ¬fa, tr, ¬fa,... ∧, ∨).
By now substituting tr for ¬fa we get E (tr, tr, tr, tr, tr,... ∧, ∨) which represents a sentence with the value true; thus E cannot be a contradiction.

Proven by J.M.A. Lutgens, R.P. Nederpelt Ph.D. (Eindhoven University of Technology, department of mathematics), because this could not be found in the literature.

Theorem 3 can be extremely useful because by using it, a quick inspection of the sentence will reveal whether or not it can be a tautology or contradiction.

Example :

(X ∧ ¬ Y ∧ Z) ∨ U → cannot be a tautology or contradiction because none of the propositions occurs in a positive as well as in a negative sense

(X ∧ Y ∧ ¬ X) → can be a tautology or contradiction because the proposition X occurs in a positive as well as in a negative sense (This sentence is a contradiction).

Theorem 4 : If a sentence X is a tautology then ¬ X is a contradiction.
(Proof by definition).

5.3 The semi-symbolic evaluator

The theorems of proportional calculus, as described in the previous section, provide us with a mathematical foundation for checking the logic of the rule base. Besides this, another important mechanism is needed for checking the rule base : a semi-symbolic evaluator [Quine, 1958]. The semi-symbolic evaluator can best be explained by giving an example :

Assume that a rule has an evaluation expression '*(X and Y) or Z*'. Also assume that we know that the value of *Z* is always TR. Then we can enter this knowledge into the expression by writing it as '*(X and Y) or TR*'. Such an expression that includes at least one constant value (TR or FA) is called a semi-symbolic expression. A purely symbolic expression like ' *P or Q or S*' does not contain such constant values.
The semi-symbolic evaluator now tries to simplify the expression by using the following eight obvious simplification rules :

Table 5.4 Simplification rules

(TR and ANY) → ANY	**(TR or ANY) → TR**		
(ANY and TR) → ANY	**(ANY or TR) → TR**		
(ANY and FA) → FA	**(ANY or FA) → ANY**		
(FA and ANY) → FA	**(FA or ANY) → ANY**		

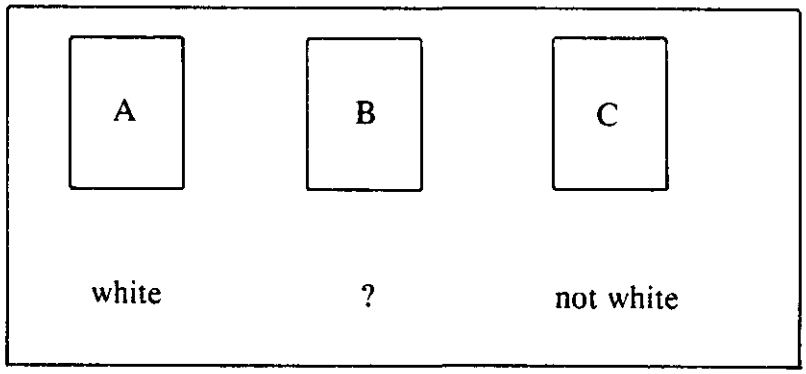
Where ANY is any expression

By using these simplification rules, we immediately see that the expression '*(X and Y) or TR*' reduces to *TR*. The advantage of semi-symbolic evaluation is clear; we do not have to know the values of *X* and *Y* to assign a value to the expression. The semi-symbolic evaluator *logically* reduces the expression.

By using a symbolic evaluator it is possible to *reason with incomplete knowledge*. A good example of this is given to me by J.A. Blom Ph.D. (Eindhoven University of Technology, department of Electrical Engineering) :

We have three boxes, A, B, and C. Box A is known to be white, box C is known to be not white. The color of box B is unknown. The question is :

Is there a white box next to a non-white box ?



After a little thought it is clear that the answer is yes.
If box B is white, then B and C are the white and the not-white boxes next to each other. If box B is not-white, then A and B are the white and not-white boxes next to each other.
In order to obtain this answer, some reasoning is required. This reasoning process can also be done by a symbolic evaluator :

Define the following primitives :

A = = box A is white, value = true.
B = = box B is white, value = unknown.
C = = box C is white, value = false.

The problem is then :

(A and not B) or (not A and B) or (B and not C) or (not B and C)

Naive insertion of the values leads to

(TR and not ?) or (not TR and ?) or (? and not FA) or (not ? and FA) =

(? or FA or ? or FA) = ? or ? = ? (unknown)

Symbolic evaluation leads to

(TR and not B) or (not TR and B) or (B and not FA) or (not B and FA) =

(not B or FA or B or FA) = not B or B = TR (true)

We can see that symbolic evaluation leads to the correct answer, whereas naive insertion does not.
By using symbolic evaluation it is possible to make logical deductions, and that is just what we need for checking the rule base. Using the symbolic evaluator in the semantic checker, the checker '*understands*' logic.

5.3.1 Quine's method

The semi-symbolic evaluator is a valuable tool for simplifying semi-symbolic expressions. In the SIMPLEXYS rule base, however, semi-symbolic expressions do not occur, only purely symbolic expressions.
The question now is 'How can the semi-symbolic evaluator be used for simplifying purely symbolic expressions '?

i.e. say that we want to know if the symbolic rule (or expression) :

(notQ and R) or (P and Q) or (notR and P) or (notP and R) or (notP and S) or (notR and notS)

is a tautology or not.

One way to examine this is by simply inserting all possible combinations of propositions. If the outcome of the expression is *always* true then the expression is considered a tautology.

Table 5.5 Insertion of all possible values

P	Q	R	S	Expression
T	T	T	T	T
T	T	T	F	T
T	T	F	T	T
T	T	F	F	T
T	F	T	T	T
T	F	T	F	T
T	F	F	T	T
T	F	F	F	T
F	T	T	T	T
F	T	T	F	T
F	T	F	T	T
F	T	F	F	T
F	F	T	T	T
F	F	T	F	T
F	F	F	T	T
F	F	F	F	T

Conclusion : the expression is a tautology.
This method is, however, impractical. The number of combinations which have to be inserted is exponentially related to the number of propositions in the expression (2^n).
If e.g. the expression contains 30 propositions (which is not unlikely in a rule base), then we would have to check $2^{30} \approx 1$ billion combinations. If we need 0.01 second to insert a certain combination and evaluate the expression , then it would take 125 days to examine all combinations.

A better way to check for tautologies is by using Quine's method [Quine, 1958]. This method implements the following idea :

* Select a proposition that occurs in the expression.
* Replace that proposition by the value TR, use the semi-symbolic evaluator to simplify the expression and note the resulting sub-expression.
* Replace the same proposition by the value FA, use the semi-symbolic evaluator to simplify the expression and note the resulting sub-expression.
* Repeat this until all sub-expressions are propositions themselves or constant values (TR or FA).

The following example demonstrates Quine's method :

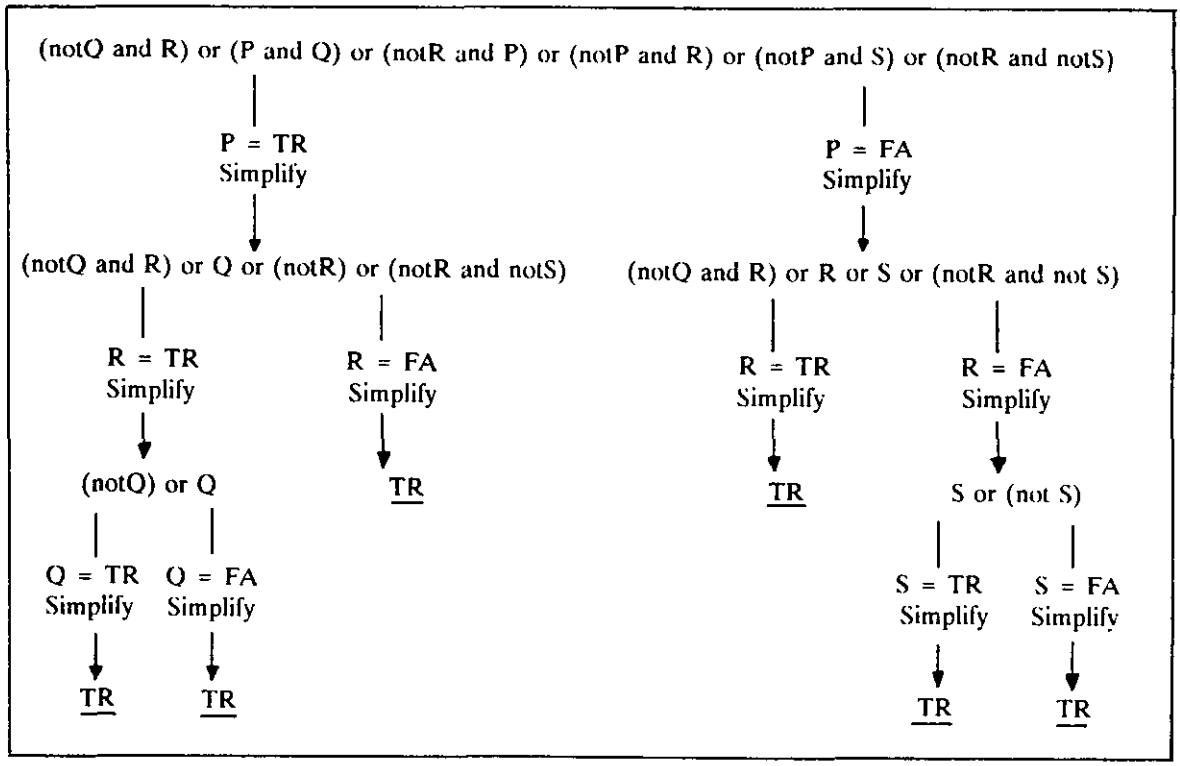


Figure 5.1 Evaluation according to Quine's method

We can see that all outcomes are TR , so the expression is a tautology.
By using Quine's method the number of steps which have to be taken is limited.

In the example above the number of possible outcomes reduced from the usual 2^4 (by simple insertion) to only 6 (these are underlined in figure 5.1). By using the semi-symbolic evaluator after each replacement, part of the expression disappears, thus limiting the number of outcomes. This becomes even more important for larger expression (more will disappear after substituting an proposition).
A good substitution strategy is to choose the proposition that has the greatest number of repetitions and also occurs in a positive as well as in a negative sense; this strategy tends to hasten the disappearance of propositions and thus to minimize the work. Furthermore the process can stop without finding a tautology as soon as one of the sub-expressions evaluates to a constant value that differs from earlier found values or when a sub-expression is found that cannot possibly represent a tautology (theorem 3).
Quine's method is very useful; although simplifying the expression takes more time, this method is much faster than checking for all combination (for checking a medium rule base on the average 10000 times faster).

5.4 The semantic checker

Given the three new tools (*Theorems in propositional logic, the semi-symbolic evaluator and Quine's method*), we can now build a semantic checker that checks the knowledge base for logical completeness and correctness.
As input the checker only needs the rinfo.qqq file, which contains almost all the arrays and tables representing the rules and their mutual connectivity.
In the rule base we can distinguish two kinds of rules : *evaluation rules and primitive rules.*
The checker must necessarily assume that all primitives are semantically independent. It is therefore recommended to structure the knowledge base in such a way, that primitives are indeed independent, so that the relations between rules become visible for the checker.
Furthermore the checker will only check the symbolic code, not the Pascal code of the rule base.

5.4.1 The connectivity matrix

To facilitate checking of the rule base, useful information about the rules is stored into a separate matrix : *the connectivity matrix [N x N]* (N = number of rules).
The connectivity matrix contains all the information about the logical network and about the thelses. The matrix is filled by using the information stored in rinfo.qqq.
Each matrix element can have one or more of the following values :

Conmatrix [i, j] =

uscode	indicates that rule i needs rule j for its evaluation.
ttcode	indicates that rule i does a then tr to rule j.
tfcode	indicates that rule i does a then fa to rule j.
tpcode	indicates that rule i does a then possible to rule j.
tgcode	indicates that rule i does a then goal to rule j.
etcode	indicates that rule i does an else tr to rule j.
efcode	indicates that rule i does an else fa to rule j.
epcode	indicates that rule i does an else possible to rule j.
egcode	indicates that rule i does an else goal to rule j.
itcode	indicates that if possible rule i then tr rule j.
ifcode	indicates that if possible rule i then fa rule j.
tpcode	indicates that if possible rule i then possible rule j.
tgcode	indicates that if possible rule i then goal rule j.

Example :

R1 : R2
R2 : R3 and R4
R3 then tr R4
R4 then fa R5

	R1	R2	R3	R4	R5
R1		us			
R2			us	us	
R3				tt	
R4					tf
R5					

CONMATRIX
us = uscode, tt = ttcode, tf= tfcode

As we can see rule R1 uses R2, and rule R2 uses R3 and R4; thus R1 also uses R3 and R4.

Expanding of the connectivities in this way is done by building the transitive closure, using the following algorithm :

```
Procedure closure;
 var i,j,k : 1..N;
 begin
  for i := 1 to N do
    for j := 1 to N do
     if connected [i, j] then
       for k:= 1 to N do
         if connected [j, k] then connected [i, k] := true;
 end;
```

Also R3 does a then tr to R4, and R4 does a then fa to R5; thus R3 does a then fa to R5.

For the thelses another (modified) transitive closure procedure is executed, which forms a combination through appropriate matching.
For more details see [Blom, 1990].

After the transitive closures the connectivity matrix is :

	R1	R2	R3	R4	R5
R1		us	us	us	
R2			us	us	
R3				tt	tf
R4					tf
R5					

Now all connections between rules are available and can be quickly accessed.

5.4.2 Checking for logical completeness

One of the aspects of checking the knowledge base is a logical check for completeness. This can be interpreted as checking for missing rules; rules which are referenced but not defined. Missing rule detection is however already covered by the *rule compiler* and thus checking for this in the semantic checker would be superfluous.

5.4.3 Checking for logical consistency

Checking for logical consistency of the knowledge base can be divided into :

- *Checking for conflict situations.*
- *Checking for redundancies.*
- *Checking for subsumptions.*

We now take a closer look how each of these three are represented in the semantic checker.

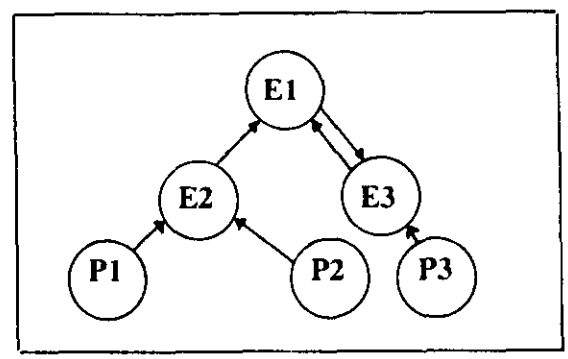
5.4.3.1 Conflict situations

By using the semi-symbolic evaluator and Quine's method, conflict detection can be expanded.
The semantic checker will check for the following conflicts :

1] Self referencing evaluation loops
An evaluation rule can get its value by means of backward chaining. The evaluation is conditional i.e. the evaluation stops as soon as the value of the expression can be calculated, but in the worst case all operands must be evaluated. An evaluation is ending if its expression is fully reducible to primitive rules. An evaluation is never ending if the rule in some way references itself. A never ending evaluation is called a loop.

An example of a loop :

E1 : E2 or E3
E2 : P1 and P2
E3 : P3 and E1
so E1 : (P1 and P2) or (P3 and E1)



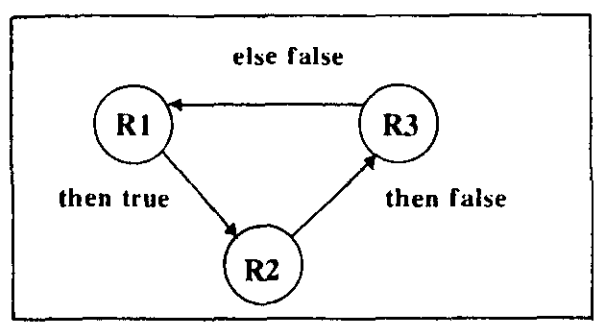
E denotes evaluation rule; P denotes primitive rule.

Evaluation loops can be detected by checking the diagonal entries of the connectivity matrix after the transitive closure has been built.
If Conmatrix [i, i] contains a *uscode* then the rule refers to itself : evaluation loop.
Loop detection was also provided for in the previous semantic checker. But it has been modified. Now the checker not only reports that a rule is part of a loop but also which loop it is a part of, by giving the loop elements. This is very useful if the rule base contains several loops.

2] thelses loop
A thelse loop will lead to a conflict if the rule that starts a chain of thelses is assigned a conflicting conclusion by that chain.

An example :

R1 then tr R2
R2 then fa R3
R3 else fa R1



This example shows an incorrect small chain that whenever rule R1 has the value TR, also tries to make it FA.
A thelse loop can also be found by inspecting the diagonal of the conmatrix.
Just as with evaluation loops, not only will be reported whether a rule is part of a

thelse loop but also which theise loop it is a part of.

3] Conflicting ALT arguments
As stated before the ALT operator is used to indicate that its two arguments are logically equivalent. The two arguments may therefore never been assigned an opposite value.

An example :

E1 : P1 and P2
E2 : P1 or not P2
E3 : E1 alt E2 ➡ E3 : (P1 and P2) alt (P1 or not P2)

If P2 is false then E3 reduces to (FA alt TR), which is of course a conflict, for the two arguments are assigned opposite values.

Checking for these conflicts is done as follows :
By definition the arguments of the ALT operator have to be logically equivalent; therefore no conflict will occur if (E1 ▪ E2) is a tautology (always true), and tautologies can be detected by using Quine's method.

For the given example this yields :

(1) (E1 ▪ E2) ➡

(2) ((P1 and P2) ▪ (P1 or not P2)) ➡ by using (3) of table 5.2

(3) (not (P1 and P2) or (P1 or not P2)) and (not (P1 or not P2) or (P1 and P2))

Applying Quine's method reveals that the latter expression is not a tautology, which means that E1 and E2 are not logically equivalent : conflicting ALT arguments.

Conflicting arguments of the ALT will be reported to the user in an appropriate way.

Once these conflicts have been removed from the rule base, the ALT can be treated as an ordinary operator like AND and OR, with its own simplification rules for semi-symbolic evaluation:

Table 5.6 Additional simplification rules of the ALT

(TR	**alt**	**ANY)**	**↔ TR**
(ANY	**alt**	**TR)**	**↔ TR**
(ANY	**alt**	**FA)**	**↔ FA**
(FA	**alt**	**ANY)**	**↔ FA**

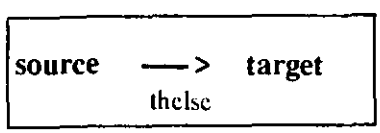
Where ANY is any expression

These extra ALT simplification rules will only be added to the simplification list if the rule base contains ALT operators.
If no ALTs are present, adding them would only unnecessarily slow down the program (extra simplification rules imply more computing time).
Conflicting ALT arguments can not be checked if both arguments are primitives.

4] Detection of thelses to successors
If a rule *needs information* from rules lower in the net, it cannot also already *know* the conclusions of those rules : thelses to successors.
Detection of thelses to successors is accomplished by inspecting the connectivity matrix. If conmatrix [i, j] contains a *uscode* and conmatrix [j, i] contains a thelse code then there is a thelse to successor : rule j uses rule i , and rule j does a thelse to rule i.

5] Detection of thelses to predecessors
If a rule *needs information* from rules below it in the net, the lower rule cannot also *know* the conclusion of the one above : thelses to predecessors.
Detection of thelses to predecessors is also accomplished by inspecting the connectivity matrix. If conmatrix [i, j] contains a uscode and conmatrix [i, j] contains a thelse code then there is a thelse to predecessor : rule j uses rule i , and rule i does a thelse to rule j.

6] Conflicting source-target thelse
Thelses are very popular in rule base construction. Depending on the outcome of a rule we may assign multiple conclusions to other rules.
The use of thelses can, however, cause inconsistency. This will be clear if we examine the construction of a thelse. A thelse is constructed as follows :

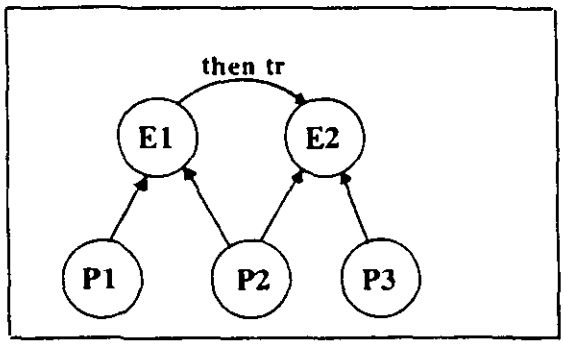


(i.e. CHILD then fa : ADULT)

Depending upon the value of the source, the target is assigned a value.

Now consider the following situation, where source and target have common variables.

E1 : P1 and P2
E2 : not P2 and P3
E1 : then tr E2



In this case E1 is the source rule and E2 is the target rule, and the primitive P2 is the common variable.
Now consider the situation where P1 and P2 are both true. If E2 is evaluated first and gets the value FA, then evaluating E1 gives an error :

E1 evaluates to TR and activates the thelse that tries to make E2 TR. This is in conflict with the value that E2 already has, FA. This type of inconsistency has to be detected.

Checking for these source-target conflicts is done as follows :

Source and target rule, each fully decomposed into primitives rules, are combined in an expression. This expression is constructed in such a way that whenever the outcome of the expression is TR, an erroneous source-target thelse is found.

The combined expression is constructed as follows :

source thelse target	combined expression
E1 then tr E2	**⇔ (R1 and not R2)**
E1 then fa E2	**⇔ (R1 and R2)**
E1 else fa E2	**⇔ (not R1 and R2)**
E1 else tr E2	**⇔ (not R1 and not R2)**

Once the combined expression has been constructed, Quine's method can be used to see if the expression can get the value TR (that is : the expression is not a contradiction).

The combined expression for the example is :

E1 and not E2 ⇔

(P1 and P2) and not (not P1 and P2 and P3) ⇔ (written out to primitives)

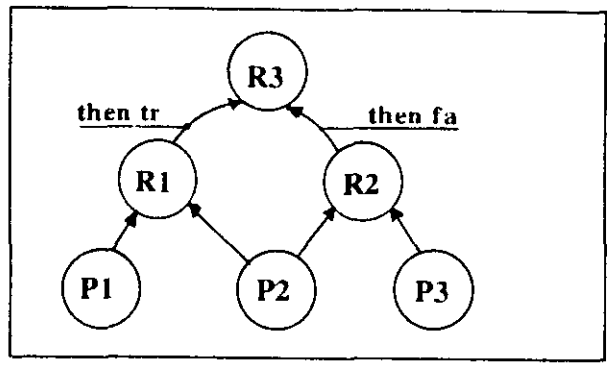
Using Quine's method will, indicate that the combined expression is not a contradiction, thus a source-target thelse conflict *can* occur.

Conflicting source-target thelses are reported to the user.

Checking is only necessary if source and target have common primitives. If all primitives are independent, this check is complete.

7] Common target conflict
Let's consider the following situation in which two different rules assign a value to a common target rule by using thelses :

R1 : P1 and P2
R2 : P2 and P3
R1 then tr R3
R2 then fa R3



If two different rules, with common variables, both have a thelse to a common target rule and if they both assign a different value to that rule then the situation has to be checked, for inconsistency may occur.

In the example a conflict will occur if both R1 and R2 are TR : common target conflict; opposite value assignment to the target rule.

The test, is again, to combine the two source rules R1 and R2 (decomposed into their primitives) into an appropriate expression and then check (by using Quine's method) whether or not that expression can get the value TR. If this is possible, then a common target conflict *can* occur.

The combined expression has to be constructed as follows :

R1 then tr R3
R2 then fa R3 → **R1 and R2**

R1 else tr R3
R2 then fa R3 → **not R1 and R2**

R1 then tr R3
R2 else fa R3 → **R1 and not R2**

R1 else tr R3
R2 else fa R3 → **not R1 and not R2**

If the combined expression is not a contradiction then a common target conflict *can* occur and this is reported to the user.

8/ Tautologies and contradictions
Tautologies and contradictions are very peculiar logical sentences, for they are always TR, respectively FA.
A rule base will normally not contain tautologies or contradictions, for it makes no sense to add a rule to the rule base that is always TR or FA.
Tautologies/contradictions will, however, not cause run time errors; they will be evaluated just as other rules. Nevertheless the rule base is checked for occurrence of tautologies and contradictions, because a tautology or contradiction usually is caused by an implementation error, such as : wrong combination of variables, misinterpretation of the expert's knowledge, typing errors .
Tautologies/contradictions can therefore be used as an indication for incorrect implementation of knowledge. In such a case the rule has to be fixed.
If, however, the tautology is not caused by an implementation error then the expression can be replaced by its constant value (TR or FA), thereby simplifying the rule base : the rule can be removed and all rules that refer to this constant rule can be simplified, or, in turn, eliminated as well.

Checking the knowledge base for contradictions/tautologies is done as follows :
First decompose the expression into its primitives.
Then apply Quine's method to verify whether or not the expression is a tautology c.q. contradiction.

Example :

R1 : P1 and P2
R2 : not P1 and P3
R3 : R1 and R2

Decompose R3 into its primitives : **(P1 and P2) and (not P1 and P3)**

Applying Quine's method will indicate that the expression is a contradiction; thus R3 is a contradiction.

It is obvious that an expression that is a tautology because of the semantic of the *primitives* will not be found.

Example :

P1 : BTEST length > 180
P2 : BTEST length < = 180

R3 : P1 or P2 ⇒ R3 : BTEST length > 180 or BTEST < = 180

R3 is a tautology, but this will not be detected because the Pascal code of the test rules is not available to the checker and therefore the checker must necessarily assume that all primitives (including P1 and P2) are independent.

To allow more comprehensive checking, the knowledge engineer should remove P2 and replace it by *not P1*, thus making the tautology visible to the checker.

9] Partial tautologies
A tautology implies that an expression is independent of all its variables. A more frequent problem, however, is that through improper combinations of rules, a rule becomes independent of one or more of its variables, but not of all of them.

For example :

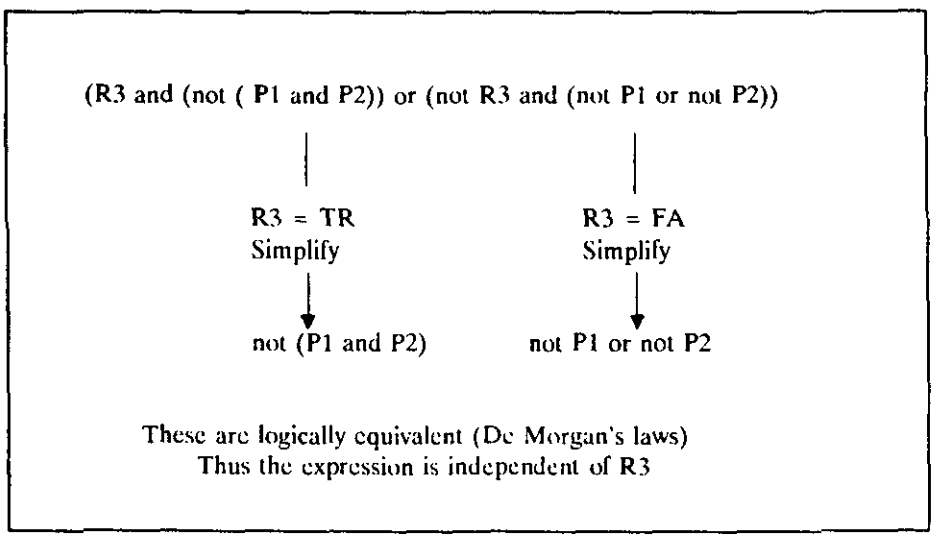
(P1 and (P2 and not P2)) can be reduced to P1

(P3 or (P3 and P4)) can be reduced to P3

Whenever a variable can be removed from the expression we say that the expression contains a *partial tautology*.

A partial tautology indicates that the expression is independent of one or more of its variables. This implies that substituting TR for the variable in question, or substituting FA for the variable in question yields the same sub-expression.

Example :



Partial tautologies are detected as follows :

Choose one variable and make two sub- expressions, one with the value TR substituted for that variable and one with the value FA. The expression is now

independent of that variable if the sub-expressions are logically equivalent.

Logical equivalence can be tested by examining whether or not the expression

(Sub-expression 1 = Sub-expression 2)

is a tautology.

If the expression is a tautology then the variable can be eliminated.
Partial tautologies occur even when a term does not occur in both positive and negative sense in the expression (i.e. P1 and (P2 and P1) = P1).
This makes finding partial tautologies more time consuming then finding tautologies (we can not rule out an expression beforehand as was the case with tautologies by using theorem 3).

10] Unconnected rules
Sometimes a knowledge engineer defines a rule, but simply forgets to use it. Such rules are unconnected. The checker has to remind the knowledge engineer that the rule is unused, and therefore it has to be able to detect these unconnected rules.

Rules in SIMPLEXYS are unconnected when :

* The rule is not used in any expression.
* The rule is not thelsed by another rule.
* The rule is not a trigger rule.

Unconnected rules can be found by inspecting the connectivity matrix; if row i of the connectivity matrix is empty then rule i is unconnected.

11] Inference addition
As opposed to boolean logic, SIMPLEXYS uses a third value : PO (possible, unknown). By using the value PO we can reason with incomplete knowledge (see the example of the three boxes).
The value PO can, however, cause undesirable behaviour of the system.
This can be best illustrated by an example :

Define the following rule : You will keep dry if it does not rain or it rains but you have an umbrella.

Converting this rule into formal logic yields :

D : not R or (R and U)

where D = 'you will keep dry'; R = 'it rains'; U = 'umbrella'

Now consider the following question :

I do not know whether or not it rains (R=PO), but I have got an umbrella (U=TR); will I keep dry ?

The answer is of course 'yes', because you will always keep dry when you have an umbrella.

Solving this question by symbolic evaluation yields :

D: not R or (R and TR) → not R and R → TR

The inference engine of SIMPLEXYS, however, cannot handle this expression properly :

D: not R or (R and TR) → not R and R → not PO and PO → PO

The inference engine comes up with an incorrect answer. This problem occurs because the *PO* and the *not PO* are correlated, which the inference engine does not know.

The same holds for the expression '**F : X and (not X or Y)**' which is evaluated incorrectly by the inference engine if X is PO and Y is FA.

Note that the problem can only occur when a variable occurs in a positive as well as in a negative sense *and* the value PO is substituted for the variable.

The problem is that the inference engine incorrectly evaluates the following expressions when the value PO is substituted for this variable :

P and not P → FA (inference engine : PO and not PO → PO);

P or not P → TR (inference engine : PO or not PO → PO);

This discrepancy is unsatisfying and must be fixed.
What we need to see is that the inference engine gives the same results as symbolic evaluation would.

This can be achieved as follows :

If an expression references a term 'P', both in a positive and in a negative sense then we can always rewrite the expression as

E : (P and E1) or (not P and E2) or ((P and not P) and E3) or E4

or by using shorthand notation : $E = P \cdot E1 + \overline{P} \cdot E2 + P\overline{P} \cdot E3 + E4$

The inference engine will, however, evaluate this expression erroneous whenever this expression reduces to *'(P and not P)'* or *'(P or not P)'* and P has the value PO.

The inference engine will evaluate the expression correctly for *all* values of P, if the expression is expanded, resulting in the following new expression (E^*) :

E^* : (E or (E1 and E2)) and (E1 or E2)

Logically, the expansion is superfluous, but with it, the inference engine evaluation proceeds correctly for *all* values of P. If the expression contains more terms in both a positive and a negative sense, extra expansions for these terms must be introduced as well. With this procedure, a correct result is obtained in all cases.

Applying this for the two examples given above, yields :

1] D : not R or (R and U) ⇒ rewrite in the form given above.
D : (R and U) or not R → expanding → D^* : D or U

If now R = PO and U = TR are substituted, then the inference engine *will give* the correct solution :

D^* : D or U = PO or TR = TR

2] F : X and (not X or Y) ⇒ rewrite in the form given above
F : (X and Y) or (X and not X) → expanding → F^* : F and Y

If now X = PO and Y = FA are substituted, then the inference engine will give the correct solution :

F^* : F and Y = PO and FA = FA

Note that expansion is only necessary if an expression contains a term in a negative as well as in a positive sense and this variable can have the value PO.

Replacing E by E^* whenever needed, solves the problem of incorrect evaluation by the inference engine.
Expanding the expression, however, is only possible if the two sub-expressions E1 and E2 are known. General methods for obtaining these sub-expressions are not available.

The semi-symbolic evaluator can however be used to obtain two sub-expressions which can replace E1 and E2. This is done as follows :

Given the general form of an expression

$$E = P \cdot E1 + \bar{P} \cdot E2 + P \cdot \bar{P} \cdot E3 + E4$$

Set P to TR and substitute this in the expression.
Then simplify the rule by using the semi-symbolic evaluator, yielding the first sub-expression (PosExpr), which can be used instead of E1 :

$$PosExpr = TR \cdot E1 + FA \cdot E2 + TR \cdot FA \cdot E3 + E4 = E1 + E4$$

Set P to FA and substitute this in the expression.
Then simplify the rule by using the semi-symbolic evaluator, yielding the first sub-expression (NegExpr), which can be used instead of E2 :

$$NegExpr = FA \cdot E1 + TR \cdot E2 + FA \cdot TR \cdot E3 + E4 = E2 + E4$$

We can now use *PosExpr* and *NegExpr*, instead of E1 and E2 to expand the expression :

E^* : (E or (PosExpr and NegExpr)) and (PosExpr or NegExpr)

This expansion is logically the same as the one discussed before. Although PosExpr and NegExpr cause some computational overhead (due to the extra factor E4), they can equally well be used for expanding an expression.
The great advantage of PosExpr and NegExpr is that they can be obtained easily.

Whenever an expression needs an expansion, to make sure that the inference engine will evaluate it correctly, the semantic checker will report the two sub-expressions, PosExpr and NegExpr to the user. The user should then replace the expression by :

E^* : (E or (PosExpr and NegExpr)) and (PosExpr or NegExpr)

thus yielding correct inference engine evaluation in all cases.

The semantic checker thus solves an inferencing problem which cannot be solved by the inference engine itself.

5.4.3.2 Redundancy

Redundancy means that a part of the rule base is superfluous.
As opposed to conflicts, redundancies will not lead to run time errors or incorrect inferencing of the inference engine. Nevertheless we check for redundancy, because redundancy usually indicates an implementation error. If not, redundant

rules can be removed from the rule base without affecting the operation of the expert system; the effect is a more compact rule base, and thus a more efficient evaluation.

The semantic checker checks for two kinds of redundancy in the SIMPLEXYS rule base :

1] FACT rule reduction

2] Equivalent rules

FACT rule reduction
FACT rules are primitive rules which have a constant value (TR, PO, FA) during the expert system's run. The value of the FACT rule is obtained before the system starts up. Whenever an evaluation rule uses a FACT rule, the evaluation rule can be simplified by using the semi-symbolic evaluator, thus simplifying the rule base.

Example :

F1 : 'Fact rule number 1'
FACT

E : P1 and P2 and (P3 or P4) and F1 and P5

Define F1 = FA ; Semi-symbolic evaluation of E yields

E : P1 and P2 and (P3 or P4) and FA and P5 → FA

The expression E is now reduced to FA by substituting the FACT rule and applying semi-symbolic evaluation. Rule E is now effectively a FACT rule as well, and can be eliminated, too.

Equivalent rules
A rule in the rule base can be superfluous because it is equivalent to another rule. Equivalent rules can be easily detected if they are textually the same.

Example :

E1 : P1 and P2
E2 : P1 and P2

We see that both E1 and E2 have the same evaluation expression and are thus equivalent.
It gets more complicated when the rules are not textually the same but logically equal. This should also be reported by the semantic checker for their truth tables are exactly the same.

Two rules 'R1' and 'R2' are logically equal if (R1 ▪ R2) is always true (tautology). Logically equivalent rules can be detected by first combining the two rules in an expression, (R1 ▪ R2), and then apply Quine's method to it to see whether or not the combined expression is a tautology.

Example :

E3 : not (P1 and P2)
E4 : not P1 or not P2

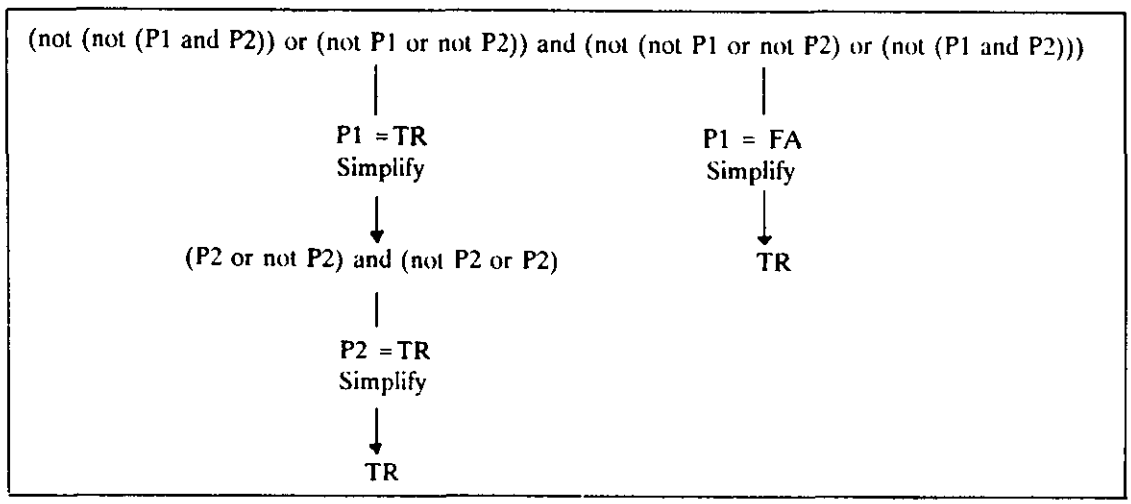
combined expression :

(E3 ▪ E4) ➡ Using (3) of table 5.2

(not E3 or E4) and (not E4 or E3) ➡ Decomposed in primitives

(not (not (P1 and P2)) or (not P1 or not P2)) and (not (not P1 or not P2) or (not (P1 and P2)))

Now applying Quine's method to see whether or not this expression is an tautology



The combined expression is a tautology; thus E3 and E4 are logically equivalent.

5.4.3.3 Subsumption

Subsumption means that some rule is a special case of another rule (like square is a special case of Rectangle; thus when square is TR then rectangle is also TR). Subsumption is similar to partial redundancy. Just as with redundancy, subsumption does not cause run-time errors. Nevertheless it may be interesting to find rules that subsume other rules, because subsumptions define some sort of hierarchy in rules. Errors in the hierarchy can be easily detected by inspecting subsumptions.

Subsumption is the same as 'logically implies' . If rule X implies rule Y then rule X also subsumes rule Y.
Subsumptions can be detected as follows :

If X subsumes Y then X implies Y (which is the same thing) is always true; thus (X - Y) is a tautology.

Example :

E1 : P1 and P2 and P3
E2 : P1 and P2

It is obvious that E1 is a special case of E2. This means that E1 subsumes E2 and that (E1 - E2) must be a tautology.

(E1 - E2) ➡ using table 5.2 (1)

(not E1 or E2) ➡ decompose into primitives

(not (P1 and P2 and P3) or (P1 and P2)

Applying Quine's method to see if the expression is an tautology is indeed affirmative; thus E1 subsumes E2.

5.4.4. Limitations

A major limitation to checking is the fact, that the checker has no access to the Pascal code. The checker therefore necessarily assumes that all primitives are independent. For history rules the same holds, the Pascal code cannot be accessed. History operators operationg on the same rule are, however, somehow dependent. If we consider all history rules as being independent then some tautologies will not be detected :

(R1 > (3)) or not (R1 >= (3))

If all histories were replaced by the rule they operate on, then the tautology becomes visible for the checker for it now sees :

R1 or not R1.

The following rule will then also be detected as an tautology, which it is not.

(R1 >10) or (not R1 <10))

We thus have the choice of checking too little, or checking too much.
The semantic checker chooses the last option, because in practice the number of errors reported due to 'over-detection' is limited.

5.5 Conclusions

SIMPLEXYS allows better consistency checks than many other system languages. With regard to the semantic checker, we may state that checks have been expanded in such a way, that it is now possible to systematically check the knowledge base for logical inconsistency and completeness.
Furthermore the different kinds of checks (conflicts, redundancy, subsumptions) are formulated in such a way that we only have to check for tautologies or contradictions.
This can be done almost linear in time by using Quine's method.
The semantic checker has now become one of the most valuable tools in knowledge base debugging.

6 Implementation of the semantic checker

6.1 Introduction

In this chapter we will discuss the implementation of the semantic checker, as described in chapter 5. The treatment of the implementation in this chapter will be global, details are left out in order to not confuse the reader, for the semantic checker is a rather complex program. We will discuss the following three items :

1] An abstract data structure for representing and manipulating rules and expressions.

2] The implementation of the semi-symbolic evaluator, used for simplifying expressions.

3] The implementation of the method of Quine.

6.2 The abstract data structure Expression

Intuitively, of course, a logical expression is a collections of propositions or variables connected by operators, such as AND, OR, NOT, ALT. The structure of a logical expression or rule is stored in rinfo.qqq in a prefix form. The prefix form enables us to easily decompose each rule into its primitives, by simple substitution.

Example :

E1 : and P1 P2 **(infix form : E1 : P1 and P2)**
E2 : or E1 P3 **(infix form : E2 : E1 or P3)**

E2 can now by decomposed into its primitives rules by direct substitution of E1.

E2 : or and P1 P2 P3 **(infix form : E2 : (P1 and P2) or P3)**

A rule or logical expression will be represented by a tree structure. Every element of the tree is of the type *Expression*. The type expression contains a pointer to a record which describes the properties of that particular junction.

```
Expression = ^ Exprstruc
 Exprstruc = record
               case kind : kindnames of
               Vbl       : (v : variable );
               Cnst      : (c : constant );
               Cmpst     : (op :operator; opnd 1,opnd2 :expression);
               NotCmpst: (op :operator; opnd : expression );
             end;
```

Vbl : Variable or proposition in a rule
Cnst : Constant value, TR or FA
Cmpst : Composition, composed of a operator , and two operands
NotCmpst : Not composition composed of an operator and an operand.

How rules are represented using the tree structure will be explained by the following example :

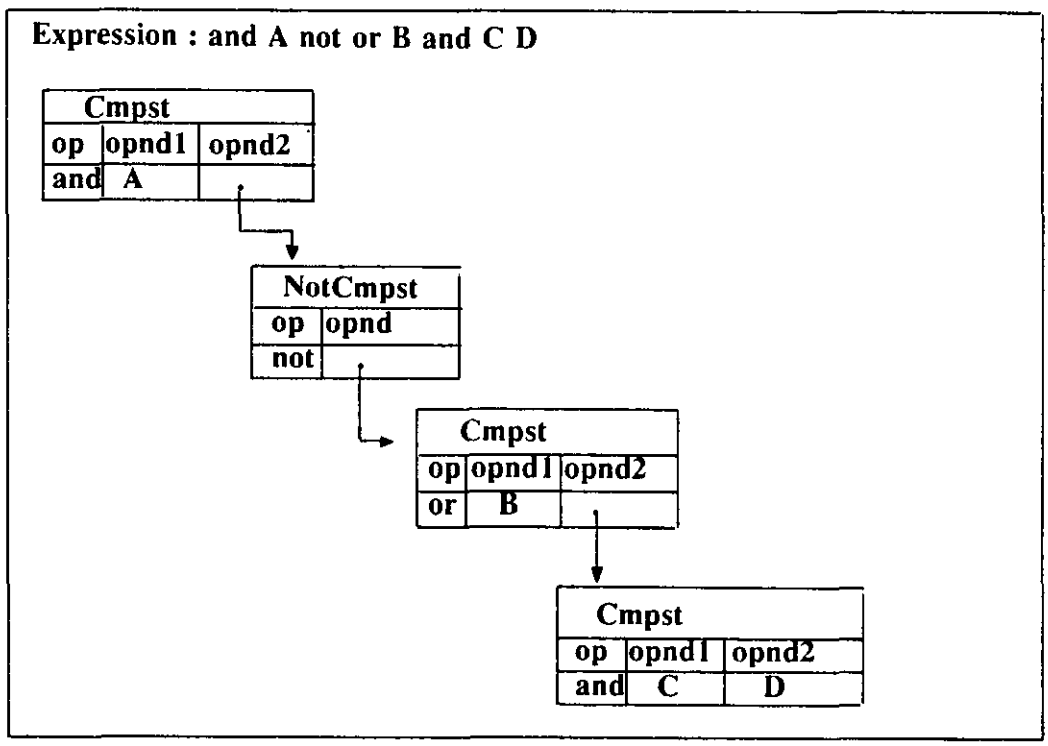


Figure 6.1 Data representation of an expression

Each rule in the rule base is decomposed into its primitive rules and represented by such a tree structure.

6.3 The semi-symbolic evaluator

The semi-symbolic evaluator is used to simplify semi-symbolic expressions. This is done by function ApplyProductions, in a recursive way [Wulf, 1981].

```
function ApplyProductions ( E :Expression; L : RefListofProductions :Expression);

 pre : The expression E is a semi-symbolic expression and can be simplified by using
 the production list of simplification rules

 post: The expression E is simplified using pattern matching.

 var Changed : Boolean; NewExpr,PrevExpr : Expression
     RemainingingProds :RefListofProduction; P :Production

 begin
  NewExpr := Copy ( E);
  repeat
   Changed := false;
   if Kind (NewExpr) = Cmpst
    then begin
      PrevExpr := NewExpr;
      NewExpr  := ConsExpr (Oper (NewExpr),
                ApplyProductions (Operand1 (NewExpr),L);
                ApplyProductions (Operand2 (NewExpr),L);
      Release (PrevExpr);
    end;
  RemainingProds := L;
  while not IsEmpty (RemainingProds) do
    begin
     FirstItem (RemainingProds,P);
     RemainingProds := Tail (RemainingProds);
     if Match (NewExpr,P.Patn) then
      begin
       PrevExpr := NewExpr;
       NewExpr := Replace (NewExpr, P.Repl);
       Release (PrevExpr);
       Changed := True; RemainingProds := L;
      end;
     end;
  until not Changed;
  ApplyProductions := NewExpr;
 end;
```

The simplification is done by traversing the tree structure of a rule until the leaves are reached. Starting from the leaves the tree is traversed in reverse order, trying to simplify each sub-expression it encounters by using the simplification rules (table 5.4).
If the sub-expression matches one of the patterns of the simplification rules then the sub-expression is simplified by substituting the replacement for it.
Pattern matching is only possible if the sub-expression is semi-symbolic. This is done until the root of the tree is reached.

The simplification rules are stored in a linked list. Each element of the link list contains a record composed of a pattern and a replacement. The link list can be easily expanded.

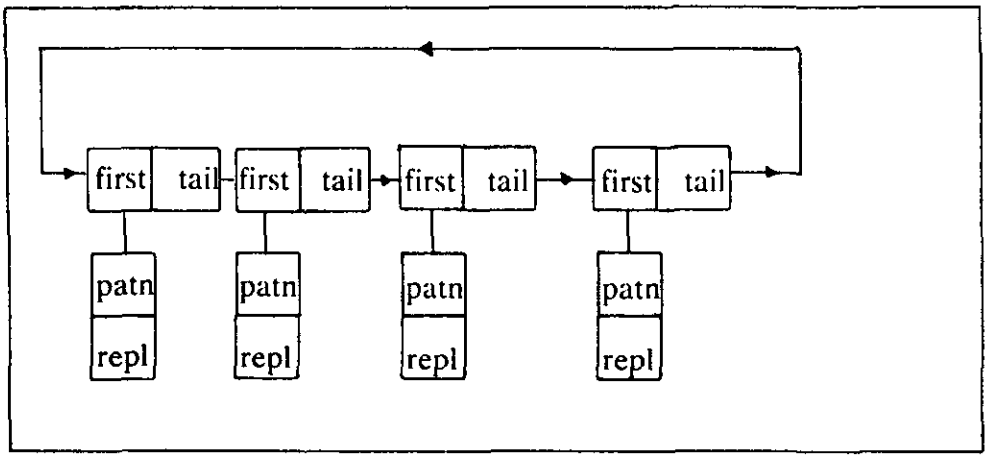


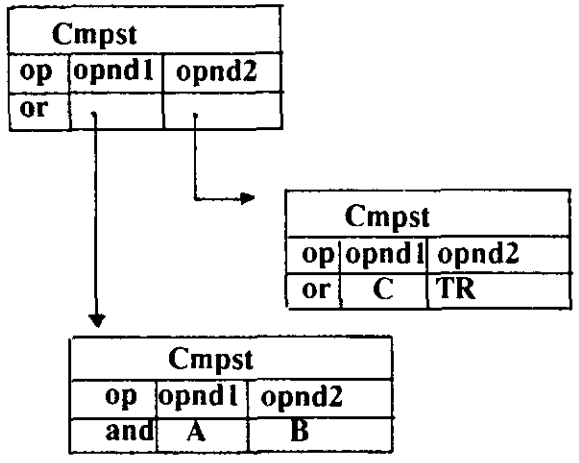
Figure 6.2 Linked list of simplification rules

Example :

Expression : (A and B) or (C or D) with D = TR

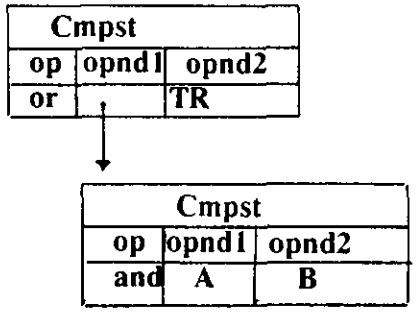
prefix form : or and A B or C D

data structure :



Sub-expression C or TR can be simplified by using ANY or TR → TR

New tree structure :



Sub-expression (A and B) or TR can be simplified by using ANY or TR - TR

Conclusion : the expression is TR.

6.4 The method of Quine

Quine's method used for simplifying purely symbolic expressions is implemented as follows :

```
Procedure Quine (Expr :Expression; var Res :Range; var Red : boolean; var Teller : integer );

 var PosExpr,NegExpr: expression;
          Nextvar    : integer;
          Ontk       : boolean;

 begin
  if (Res < > Stop) and (not Red) then begin
    case Kind (expr) of
      Cnst : Begin
               Teller := Teller-1;
               Res := Check (Expr,Res);
             end;
      Vbl : Res := Stop;
      else begin
           Ontk := false;
           ClearAdm (Ontk);
           ChangeAdm (Expr,Ontk);
           Nextvar := Findvar;
           if Nextvar = empty then Res := Stop;
           else begin
              NegExpr := Expr;PosExpr := Expr;
              Teller := Teller +2;
              MakeNeg (Nextvar);
              NegExpr:= ApplyProductions(NegExpr,Simplist);
              MakePos (Nextvar);
              PosExpr := ApplyProductions(PosExpr,Simplist);
              Remake (Nextvar);
              Quine (Negexpr, Res, Red, Teller);
              Quine (PosExpr, Res, Red, Teller);
           end;
       end;
    end;
 end;
 if Res =Stop then Red := true;
 if Teller < =0 then Red := true;
end;
```

Quine's method is implemented as follows :

If the expression is a composition then a variable is selected for substitution (the variable has to be present in positive as well as negative sense and it is the variable which occurs the most in the expression). Substituting this variable by making it TR yields the PosExpr; Making it FA yields the NegExpr. These two sub-expressions are then simplified. Each sub-expression is then tested again if it is a tautology or contradiction. This recursive process ends when a sub-expression reduces to a variable or a constant value different from the one before.

6.5 Conclusion

The semantic checker has been successfully implemented. It is easy to use and menu-driven. Several test cases have been applied to the checker. All errors in the test-cases were detected by the semantic checker. Systematically checking for logical completeness and consistency of the rule base is now possible. During development the semantic checker has been used by several people and all of them agreed that the semantic checker is a valuable tool for knowledge base debugging.
Furthermore the semantic checker produces an error file with all the errors in the rule base summarised in a proper form, which can also be sent to the printer.

7 Conclusions and future work

Knowledge base debugging is one of the hardest task in expert system development. Therefore any tool which in some way contributes to the correctness of the rule base is welcome. The semantic checker, designed for checking the logical completeness and consistency of the rule base is, however, much more than an ordinary tool. It systematically checks the rule base for certain errors, thus dramatically improving the confidence in the correctness of the knowledge base. The semantic checker is easy to use and sufficiently fast. It takes only 5 minutes to check a rule base of approximately 180 rules. Numerous test cases have been applied and all errors were detected by the semantic checker.
Furthermore the semantic checker provides the user with a extensive error file, needed to directly trace the bugs and fix them.
Many users have experimented with the checker and all agreed that it is a very valuable tool for expert system debugging.
Nevertheless the semantic checker can be improved. One idea is to automatically change the rinfo.qqq file whenever a bug is detected; thus yielding bug repair without intervention of the user. Of course the user will always be informed about the changes.
A second idea is to integrate the semantic checker with the protocol checker. The protocol checker is thus provided with an option for testing the logical equivalence of triggers and the semantic checker thus can analyze certain rules with regard to their context. Integration has already been partially realized and appears to work perfectly.
The best improvement can be achieved by revealing the Pascal code to the checker, but for this the rule compiler should be equipped with a Pascal compiler as well.
A major effort of the checker is that all errors can be converted in such a way that we only have to check for tautologies or contradictions. By using this we also capture the value PO, because a contradiction (or tautology) is always a contradiction (or tautology) even if PO is substituted for the value of the variables.
The semantic checker currently can handle only rule bases up to approximately 200 rules. This is not caused by the checker itself but due to the limitations of Turbo Pascal. The number of rules is restricted because the connectivity matrix may not exceed 64K (largest single structure on the heap). Expanding the 64K limit in Turbo Pascal can be done but then memory has to be addressed directly which is not recommended. This limitation has not yet caused any trouble because no rule base having more then 200 rules has been built yet.
We may state that the semantic checker has become a very valuable debugging tool; a step forward in knowledge base debugging.

References

Blom, J.A.
The SIMPLEXYS experiment: Real time expert systems in patient monitoring.
Ph.D. thesis. Eindhoven University of Technology, 1990.

Boon, P.M.C.
Efficiëntie en correctheid van SIMPLEXYS expert systems (in Dutch).
M. Sc. thesis. Division of Medical Electrical Engineering, Faculty of Electrical Engineering, Eindhoven University of Technology, 1987.

Feigenbaum, E.A.
Expert systems in the 1980s.
In: State of the art review 1980. Proc. day 3.
Maidenhead, Berksh.: Infotech, 1980. P. 67-77.

Frenzel, Jr., L.E.
Crash course in artificial intelligence and expert systems.
Indianapolis, Ind.: Howard W. Sams, 1987.

Hasling, D.W. and W.J. Clancey, G. Rennels
Strategic explanations for a diagnostic consultation system.
Int. J. Man-Mach. Stud., Vol. 20, No. 1(1984), p. 3-19. Reprinted in: Development in expert systems. Ed. by M.J. Coombs. London: Academic Press, 1984. Computers and people series. P. 117-133.

Lammers, J.O.
The use of Petri net theory for SIMPLEXYS expert systems protocol checking.
Faculty of Electrical Engineering, Eindhoven University of Technology, 1990.
EUT Report 90-E-238

Lutgens, J.M.A.
Een tautologie-checker voor het SIMPLEXYS expert systeem (in Dutch).
Stageverslag. Division of Medical Electrical Engineering, Faculty of Electrical Engineering, Eindhoven University of Technology, 1989.

Mitchie, D.
The state of the art in machine learning.
In: Introductory reading in expert systems. Ed. by D. Mitchie. London: Gordon & Breach, 1982. Studies in cybernetics, Vol. 1. P. 208-229.

Philippens, E.H.J.
Designing debugging tools for SIMPLEXYS expert systems.
Faculty of Electrical Engineering, Eindhoven University of Technology, 1990.
EUT Report 90-E-234

Osterweil, I.
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.

Quine, W. Van Orman
Methods of logic.
London: Routledge & Kegan Paul, 1958.

Wulf, W.A. and M. Shaw, P.N. Hilfinger, L. Flon
Fundamental structures of computer science.
Reading, Mass.: Addison-Wesley, 1981.

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

(236) Lammers, J.O.
KNOWLEDGE BASED ADAPTIVE BLOOD PRESSURE CONTROL: A Simplexys expert system application.
EUT Report 90-E-236. 1990. ISBN 90-6144-236-2

(237) Ren Qingchang
PREDICTION ERROR METHOD FOR IDENTIFICATION OF A HEAT EXCHANGER.
EUT Report 90-E-237. 1990. ISBN 90-6144-237-0

Eindhoven University of Technology Research Reports
Faculty of Electrical Engineering

ISSN 0167-9708
Coden: TEUEDE

(238) Lammers, J.O.
THE USE OF PETRI NET THEORY FOR SIMPLEXYS EXPERT SYSTEMS PROTOCOL CHECKING.
EUT Report 90-E-238. 1990. ISBN 90-6144-238-9

(239) Wang, X.
PRELIMINARY INVESTIGATIONS ON TACTILE PERCEPTION OF GRAPHICAL PATTERNS.
EUT Report 90-E-239. 1990. ISBN 90-6144-239-7