EINDHOVEN UNIVERSITY OF TECHNOLOGY

The SIMPLEXYS experiment : real time expert systems in patient monitoring

Citation for published version (APA):
Blom, J. A. (1990). *The SIMPLEXYS experiment : real time expert systems in patient monitoring*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Electrical Engineering]. Technische Universiteit Eindhoven. https://doi.org/10.6100/IR330398

DOI:
10.6100/IR330398

Document status and date:
Published: 01/01/1990

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

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

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

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

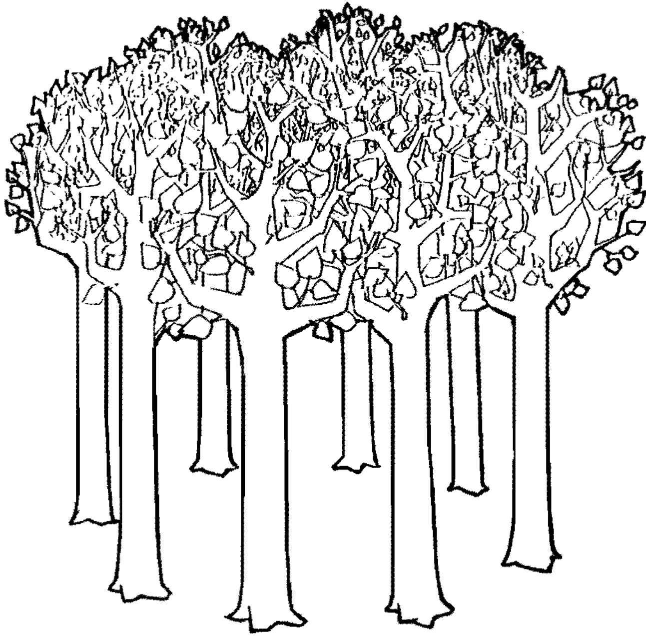
providing details and we will investigate your claim.

Download date: 07. Jul. 2024

[illegible]
17/4/90

The SIMPLEXYS Experiment

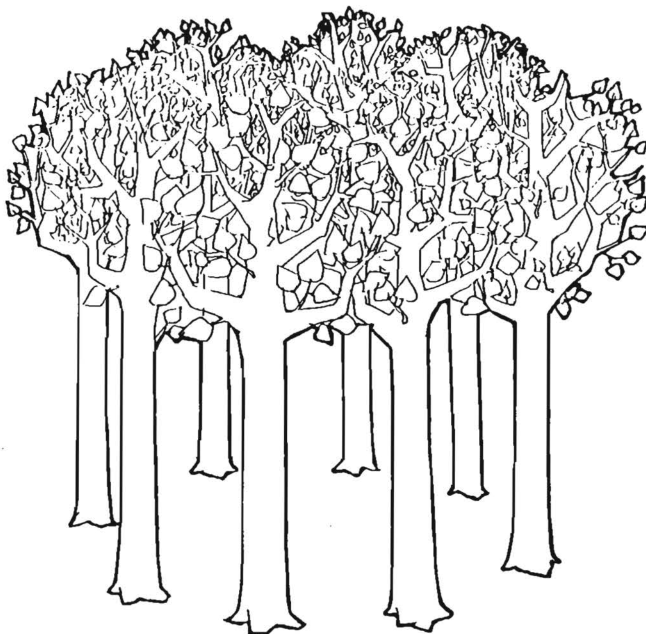
Real Time Expert Systems In Patient Monitoring



J. A. Blom

The SIMPLEXYS Experiment

Real Time Expert Systems In Patient Monitoring



J. A. Blom

The SIMPLEXYS Experiment

Real Time Expert Systems In Patient Monitoring

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van
de Rector Magnificus prof.ir. M. Tels, voor een
commissie aangewezen door het College van Dekanen
in het openbaar te verdedigen op
vrijdag 11 mei 1990 om 14.00 uur

door

Johannes Abraham Blom

geboren te Haarlem

Dit proefschrift is goedgekeurd door de promotoren

prof.dr.ir. J.E.W. Beneken

en

prof.dr. A. Hasman

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Blom, Johannes Abraham

The SIMPLEXYS experiment: real time expert systems in patient monitoring / Johannes Abraham Blom. - [S.l. : s.n.]. - Fig., tab. Proefschrift Eindhoven. - Met lit.opg., reg.
ISBN 90-9003296-7
SISO 608.1 UDC 616-089.5(043.3) NUGI 742
Trefw: anesthesie; patiëntbewaking / expertsystemen.

Learning is finding out
what you already know.
Doing is demonstrating
that you know it.
Teaching is reminding others
that they know just as well as you.

Richard Bach: Illusions

Voorwoord

Het in dit proefschrift beschreven onderzoek zou niet mogelijk geweest zijn zonder de steun van velen die op de een of andere wijze hebben gefungeerd als bron van inspiratie of, hoe dan ook, als mede-werker.

In de eerste plaats geldt dit prof.dr.ir. J.E.W. Beneken, die mij vele jaren lang heeft voorgehouden eindelijk eens een deel van mijn werk als proefschrift te bundelen. Het is er eindelijk van gekomen, Jan!

Evenzeer prof. dr. A. Hasman, wiens commentaren en kanttekeningen mij er toe brachten allerlei nodige puntjes op de i te zetten.

Daarnaast alle medewerkers van de vakgroep Medische Elektrotechniek, die het mij, door bij tijd en wijle ander werk over te nemen, mogelijk hebben gemaakt een deel van mijn tijd te besteden aan een studie van het vakgebied van de Kunstmatige Intelligentie, en in het bijzonder de Expert Systemen. Ik dank hun voor hun prettige collegialiteit.

Vervolgens alle studenten, stageairs en afstudeerders, die aan delen van dit onderzoek hebben meegewerkt, vaak met groot enthousiasme en een plezierig kritische blik. Vooral in de laatste fase is veel werk is verricht door ir. Johan Lammers. Uiteraard wordt in dit proefschrift naar het werk van al deze medewerkers gerefereerd.

Ir. Jan Hajek heeft mij op het spoor gezet van wat uiteindelijk SIMPLEXYS is geworden. Zijn zucht naar efficientie bleek besmettelijk, en zijn 'destructive mind set' meestal een stimulans om het *toch* te doen.

De medewerkers van de afdelingen Anesthesie en Cardiochirurgie van het Catharina Ziekenhuis, en in het bijzonder dr. Erik Korsten, hebben ons met groot enthousiasme en veel inzet begeleid in de klinische experimenten.

Ik denk dat ik iedereen niet beter kan danken dan door te zeggen dat ik dit onderzoek met heel veel plezier heb gedaan, dat ik veel heb geleerd, dat ik heel tevreden ben met het resultaat, en dat ik denk dat de voltooiing van dit onderzoek zal leiden tot de realisatie van een aantal buitengewoon zinvolle toepassingen.

Table of contents

1.	**Introduction**	**1**
1.1.	SIMPLEXYS: a tool to construct tools	1
1.2.	General characteristics of patient monitoring tasks	2
1.3.	General characteristics of real world information	3
1.4.	Goals of the SIMPLEXYS experiment	3
1.5.	The relevance of the SIMPLEXYS experiment	4
1.6.	Difficulties in presenting the SIMPLEXYS experiment	5
1.7.	The SIMPLEXYS toolbox	5
2.	**An introduction to expert systems**	**7**
2.1.	Introduction	7
2.1.1.	History	7
2.1.2.	Organizing knowledge	8
2.1.3.	Representing knowledge	10
2.1.4.	A critique of expert systems	14
2.2.	Real time expert systems	16
2.2.1.	Definitions of real time	16
2.2.2.	Real time expert systems are different	17
2.2.3.	Problems in real time expert systems research	19
2.2.4.	A critique of real time expert systems	20
2.3.	VM: a real time expert system	21
2.3.1.	An introduction to VM	21
2.3.2.	Knowledge acquisition	23
2.3.3.	Data and the data base	24
2.3.4.	Rules and the rule base	25
2.3.5.	Inferencing	27
2.3.6.	A review of VM	28
2.4.	The relation between expert systems and formal logic	28
2.5.	The relation between expert systems and computer science	30
3.	**The machine monitor**	**32**
3.1.	Problem solving in medicine	32
3.1.1.	Diagnostic approaches	34
3.1.2.	Protocols	35
3.1.3.	Monitoring	37
3.2.	Machine reasoning	40
3.3.	The SIMPLEXYS problem solving methodology	41
3.4.	Language design	44

4.	**The origin and evolution of SIMPLEXYS**	**47**
4.1.	Requirements for expert system applications	47
4.2.	Toward more efficiency	49
4.3.	Start of the Inference Engine	51
4.4.	Start of the SIMPLEXYS syntax	53
4.5.	Adding THELSEs	56
4.6.	Limitation to ternary logic	57
4.7.	Adding an interface with the outside world	58
4.8.	Adding forward chaining	59
4.9.	MEMO rules, STATE rules and rule histories	60
4.10.	FACT rules	65
4.11.	The relation between SIMPLEXYS and production systems	66
4.12.	A first evaluation	67
5.	**SIMPLEXYS: a real time expert systems toolbox**	**68**
5.1.	An example of a SIMPLEXYS program	68
5.2.	Elements of the SIMPLEXYS syntax	70
5.2.1.	Rule's conclusions	70
5.2.2.	Rule types	71
5.2.3.	The logic of SIMPLEXYS	72
5.2.4.	ON statements	81
5.3.	SIMPLEXYS as a programming language	81
5.3.1.	A typical operation of a SIMPLEXYS expert system	81
5.3.2.	SIMPLEXYS programs	82
5.4.	Programming in SIMPLEXYS	87
5.5.	Inferencing in SIMPLEXYS	89
5.5.1.	Single run inferencing	92
5.5.2.	Global inferencing	92
5.5.3.	The validation mode	93
5.6.	The SIMPLEXYS Toolbox	94
5.6.1.	The SIMPLEXYS Rule Compiler	95
5.6.2.	The SIMPLEXYS Options Builder	98
5.6.3.	The SIMPLEXYS Inference Engine	99
5.6.4.	The SIMPLEXYS Debugger/Tracer	99
5.7.	Worst case performance of SIMPLEXYS expert systems	105
5.8.	Summary of efficiency issues	108
6.	**Checking the semantics**	**109**
6.1.	The need for knowledge acquisition support	109
6.2.	Semantic nets: nodes and links	111
6.3.	Speed aspects	113
6.4.	Safety aspects	113
6.4.1.	Rule evaluation	113
6.4.2.	Conclusion assignment	114

6.5. Systematic checking of a knowledge base 114
6.5.1. Checking for completeness 114
6.5.2. Checking for consistency 115
6.6. Limitations 128
6.7. Conclusions 128

7. Checking the protocol 129
7.1. From ON statements to protocol 129
7.2. Petri net basics 132
7.3. Systematic checking of the protocol 134
7.3.1. Detection of syntax errors by the Rule Compiler 134
7.3.2. Detection of other syntax errors 135
7.3.3. Detection of topological errors 136
7.3.4. Detection of dynamics errors 138
7.4. Correctness checks at run time 141
7.5. Conclusions 142

8. Data and data processing 143
8.1. Data acquisition 145
8.1.1. Demographic data 145
8.1.2. Volunteered data 145
8.1.3. Discontinuous measurements 146
8.1.4. Continuous measurements 148
8.2. Feature extraction and data validation 150
8.2.1. Classification of invalid periods 159
8.2.2. Limitations of the validation algorithm 159
8.3. Analysis of features 160
8.4. The signal data base161
8.4.1. Representation of data in the data base 162
8.4.2. Necessary future research 163

9. An intelligent blood pressure controller 164
9.1. Knowledge acquisition 165
9.1.1. Controlled hypotension and sodium nitroprusside 165
9.1.2. Sodium Nitroprusside control systems 170
9.1.3. Proportional Integral Derivative controllers 172
9.2. Knowledge implementation 173
9.2.1. The arterial pressure signal 174
9.2.2. The controller 175
9.2.3. The SNP characteristics 177
9.2.4. The PID controller 177
9.2.5. Safety aspects 182

9.3. Design of an expert system based SNP controller 183
9.3.1. Data acquisition and validation 183
9.3.2. The adaptive PID-controller 184
9.3.3. The supervisor module 185
9.3.4. The user interface 187
9.3.5. Knowledge engineering properties of the system 188
9.4. Clinical tests 189
9.4.1. Data acquisition and validation performance 190
9.4.2. The control performance 190
9.4.3. The performance of the complete system 192
9.4.4. Some rule base statistics 193
9.4.5. SIMPLEXYS as a knowledge engineering tool 194
9.5. Conclusions 196

10. Discussion 198
10.1. SIMPLEXYS as a real time expert system 198
10.2. SIMPLEXYS as a programming language 199
10.3. The SIMPLEXYS toolbox 200
10.4. The blood pressure controller 201
10.5. SIMPLEXYS simplified 201
10.6. SIMPLEXYS in hardware 202
10.7. General conclusions 202

Appendix 1. The SIMPLEXYS syntax 203
Appendix 2. Additional operators 206
Appendix 3. SIMPLEXYS utilities 207
Appendix 4. The Inference Engine's main data structures 208
Appendix 5. The Inference Engine's main procedures and functions 212

References 216

Summary 231

Samenvatting 234

Curriculum Vitae 237

1. Introduction

Problem solving is the core of human activity. All of our actions are preceded by some form of problem solving process. Most problems of daily life, as well as most problems that occur during our professional activities, are solved routinely, without effort. Other problems are more difficult and require effort, both in the form of manual labor and conscious judgment.

Tools play an important role when we solve a problem, from the lowly shovel that can move more dirt than a human hand to the computer that can process numbers faster than the human mind. Tools become ever more complex. The latest assortment of tools originates in a branch of science called *Artificial Intelligence*, which aims to let machines do what earlier required human intelligence.

Although Artificial Intelligence programs are often thought of as 'brain assistants', more and more they also control physical tasks. Machine Intelligence is then not only able to *conclude*, but also to *act*. Such an 'intelligent machine' is called by a variety of names; sometimes it is called a *robot*, sometimes a *control system*. It can perceive physical facts. For example, it may have a sensor to measure a patient's blood pressure. It may also recognize that in a certain context a too high blood pressure is dangerous and should be averted. And it may also know how to start the infusion of a drug.

Artificial Intelligence depends on computers. AI researcher Donald Michie uses the term 'The Human Window', and his view of AI is that computers can be used to move an otherwise too complex or voluminous task into 'the human window' so that humans can finish the job. Thus, if we have a good understanding of what the task entails, we can attempt to build 'smart tools'.

1.1. SIMPLEXYS: a tool to construct tools

The main body of this dissertation is, however, not concerned with the analysis of a single task, although one task will be analyzed in some detail in chapter 9. It is concerned with a tool (or rather a *toolbox*, a collection of tools), which is meant to assist the design of a whole *class* of further tools to be used in various tasks in the domains of process supervision and process control, and especially in patient monitoring. Tasks in these domains are characterized by the great volume of information that must be processed, often in a short time.

With the need to handle increasing amounts of information comes an increased need for performance, in terms of availability (expert level knowledge must be accessible in situations where the continuous presence of a human expert is impossible or cannot be afforded), speed (requirements for the speed at which information must be processed have steadily

increased and will continue to do so) and correctness (correct decisions must be made at accuracies that cannot be maintained by human experts).

Decisions can only be correct if the knowledge is implemented correctly. Computer programs are, more than just concatenations of instructions to a machine, compositions of programming plans. Conventional programming languages make the instructions to the machine explicit, but they often obscure the plans, causing difficulties in program design, update or maintenance, which operate mostly at the level of changing the plans in the code. In a new programming language, plans (or protocols) merit ample attention.

1.2. General characteristics of patient monitoring tasks

The tool to be described in this dissertation, SIMPLEXYS, is thus based on a 'meta level' analysis of the features that are commonly encountered in patient monitoring tasks. Due to our experience in solving a number of patient monitoring problems [e.g. Meijler, 1986; Meijler and Beneken, 1987], such knowledge was available 'in house'. Much of this knowledge was generated by or with assistance of Electrical Engineering students as part of their M.Sc. work, and references to their theses are included where relevant.

The SIMPLEXYS toolbox is based on the discovery of the following major abstractions:

- There is always a *context* that describes which tasks need to be performed, and there is always an *event sequence* that leads to a context; such an event sequence is normally not deterministic, since it may depend on whether or when certain conditions occur.

- In each context it is possible to *specify the goals*, i.e. it is possible to state which tasks must be performed in that context.

- Many goals are *analyses* (usually of externally supplied data or internally prevailing conditions) that can be described as a *combination* of simpler analyses or lower level goals; such a combination can frequently be described in the form of a logical expression containing operators like *and* and *or*.

- All analyses finally rest on *elementary* analyses or decisions.

- The result, outcome or value of any analysis or decision can be constrained to be either *true*, *false* or *unknown*.

- The result of an analysis may resolve the outcome of one or more other analyses, eliminating the necessity to actually perform the latter.

- One or more *actions* (usually to display a result or to perform a certain procedure) may be attached to any elementary or higher level analysis; whether such actions actually need to be performed almost always depends on the outcome of the analysis.

1.3. General characteristics of real world information

The type of information that has to be handled in AI applications which deal with the real world, among them medical and monitoring applications, is of a different nature than in the exact sciences. Some of these differences are:

- No substantial part of the real world is simple enough to be comprehended and controlled without abstraction. Abstraction entails substitution of a part of the real world by a model with a manageable structure. Models thus necessarily imply a simplification. In medicine, models are pervasive. 'A diagnosis is a parsimonious description of (usually abnormal) systems performance, since in a few words it integrates a number of historical and actual observations into a relatively unified framework. It thus represents an abstraction of a disorder ...' [Attinger, 1985]. In such a description it is important to know which aspects must be considered, and which can be disregarded.

- Whereas problems in mathematics can be rigorously described, many real world processes and systems are only partially understood. Therefore only limited models of those processes are available, which cannot provide predictions under all conditions.

- The concepts used in common sense reasoning have a qualitatively approximate character; they cannot be crisply defined.

- Only partial information is available, both qualitatively and quantitatively. Not all the relevant phenomena are known, and thus not all aspects of the problem can be taken into consideration. The exactness and/or reliability of the information are also frequently unknown.

- Some problems occur so infrequently that they may not be taken into consideration, even if some signs or symptoms clearly indicate their presence.

- The goals are frequently not well-defined. A critical problem is the question what is optimal. Value judgments often replace an explicit optimality criterium.

Yet, despite the lack of 'hard' information, humans solve problems, and there is no reason to suspect that the processes are so badly understood that computers cannot be programmed to solve them, as well.

1.4. Goals of the SIMPLEXYS experiment

Just like EMYCIN [Buchanan and Shortliffe, 1984] was a tool to solve a particular class of (MYCIN-like) problems, SIMPLEXYS is a tool to solve a particular class of (monitoring-like) problems. A difference is that EMYCIN became available *after* MYCIN had stabilized, whereas SIMPLEXYS stabilized before its applications did. A common feature is, that

neither EMYCIN nor SIMPLEXYS are general purpose systems. Both are heuristic in the sense that they implement only those features necessary to solve a class of problems. For SIMPLEXYS, this has some of the same repercussions as for EMYCIN [Newell, 1984]:

'AI is both an empirical discipline and an engineering discipline. This has many consequences for its course as a science. It progresses by building systems and demonstrating their performance. From a scientific point of view, these systems are the data points out of which a cumulative body of knowledge is to develop. However, an AI system is a complex join of many mechanisms, some new, most familiar. Of necessity, on the edge of the art, systems are messy and inelegant joins - that's the nature of frontiers. It is difficult to extract from these data points the scientific increments that should be added to the cumulation. Thus, AI is case-study science with a vengeance. But if that were not enough of a problem, the payoff structure of AI permits the extraction to be put off, even to be avoided permanently. If a system performs well and breaks new ground - which can often be verified by global output measures and direct qualitative assessment - then it has justified its construction. Global conclusions, packaged as the discursive view of its designers, are often the only increments to be added to the cumulated scientific base'.

We also think that SIMPLEXYS has a similar significance [Newell, 1984]:

'The step to EMYCIN does have general significance. ... It is of a piece with the strategy of building special-purpose problem-oriented programming languages to capture a body of experience about how to solve a class of problems, a strategy common throughout computer science'.

1.5. The relevance of the SIMPLEXYS experiment

The relevance of the SIMPLEXYS experiment for problems in patient monitoring is best described by Gravenstein [1979]:

'Most preventable deaths and permanent disabilities [in anesthesia] result from the fact that the anesthesiologist did not have the information available *quickly enough* [our italics] that would have permitted timely intervention, correction of an error, treatment of a complication, adjustment of a dose or perhaps cancellation of an operation.'

But the anesthesiologist[1] is usually not to be blamed. In emergencies, he is flooded by an explosion of information and actions: a great many things need to be taken care of at the

---

[1] Bendixen and Duberman [1982]: 'The tasks of the anesthesiologist are to produce comfort, freedom from pain, and general medical management for the patient undergoing surgery and to provide good operating conditions for the surgeon to perform his tasks effectively; the anesthesiologist must provide the desired benefits for both patient and surgeon. To perform them he must efficiently minimize mortality, morbidity, and cost'.

same time and a great quantity of rapidly changing data needs to be assimilated and processed. The latter is possible only if the anesthesiologist is assisted by more and more complex data processing equipment [Blom and Beneken, 1982; Beneken and Blom, 1983], the task of which is to eliminate redundancies from the overwhelming flow of data and to present only the relevant facts. Integration of data from more than one source is important [Hengst and Krämer, 1980; van Kessel, 1981; Meijler, 1986], as well as a good man-machine interface [Coolen, 1985]. And this is where more 'intelligent' instruments become necessary. Such 'intelligent instruments' can pay the necessary continuous attention to tasks which are critical and demanding but mostly routine, both in supervision and control. They will show fast reactions. And they can also offer specialized machine 'expertise' which would not otherwise be available to the (average) clinician.

1.6. Difficulties in presenting the SIMPLEXYS experiment

No paper can fully do justice to an expert system. The SIMPLEXYS tools themselves are complex, and the subject matter of any application even more so. This makes it difficult to fully describe SIMPLEXYS even in a document of more than reasonable length, and this is even more true for any expert system designed with it. The enormous relational complexity of the system, the great number of linkages between elementary operations of the inferencing process and between pieces of knowledge, the rich structure of semantic relationships, and the complex housekeeping problems make hands-on experience while designing and/or using an expert system the best way to understand it.

Actual expert systems are, like the knowledge they implement, often 'messy', to use Newell's terminology. Because the design cycle of an expert system is often difficult and lengthy, most knowledge engineers take the attitude that 'the best is the enemy of the good' and that in the real world 'optimal' is, if not impossible, then at least irrelevant. Yet, the complexity ought to be invisible to the user, who should be offered a well-behaved and 'user-friendly' system to work with. The 'flavor' of SIMPLEXYS is thus best appreciated when actually working with either of the applications thus far built with it, the blood pressure controller described in chapter 9 or the 'intelligent alarms' system described by van der Aa [1990].

1.7. The SIMPLEXYS toolbox

The SIMPLEXYS *toolbox* consists of a collection of *tools* that assist in the design of expert systems that are compact enough to be able to run on small and inexpensive PC-like computers, and yet fast enough to handle applications like patient monitoring tasks. Chapter 2 starts with a general introduction to expert systems, then reviews an earlier system with similar characteristics, and finally focusses on the particular class of which SIMPLEXYS is a member: *real time expert systems*.

Patient monitoring tasks have certain general characteristics; these are reviewed in chapter 3. Central concepts are the *hypothesize and test* character of medical decision making and the *context dependency* of interpretations, decisions and actions.

The most important tool, the new SIMPLEXYS programming language, has two functions, just like any programming language. First, it allows certain types of human knowledge to be formally yet naturally described; this description of a body of knowledge is called the *knowledge base*. And second, it allows the computer to translate the knowledge base into an internal representation that can be easily manipulated by the computer; this translation (*compilation*) is performed by another tool, the SIMPLEXYS Rule Compiler. Combining the output of the Rule Compiler, which is the internal representation of the knowledge, with the Inference Engine, the expert system's reasoning mechanism, produces a complete expert system. Chapter 4 describes how the SIMPLEXYS language resulted from a combination of efficiency (the expert system must be fast), checkability (the expert system must be as safe as possible), and readability (knowledge base maintenance requires understandable code).

Chapter 5 subsequently gives a complete description of all the SIMPLEXYS tools: the features of the SIMPLEXYS language and how they can be used by knowledge base builders; how the Inference Engine manipulates the knowledge; and how, if errors still occur when the expert system executes, the Tracer/Debugger tool can be used to discover their cause and history.

An important task of the Rule Compiler is to detect errors in the knowledge base. *Syntactic* errors are easy to detect, since the SIMPLEXYS language fully specifies which constructs are allowed, and therefore also which are not. *Semantic* errors are errors in *meaning* and much harder to detect; yet, extensions of the Rule Compiler have the task to discover as many of these errors as possible. The Semantics Checker is described in chapter 6 and the Protocol Checker in chapter 7.

SIMPLEXYS expert systems will frequently have to analyze rapidly changing signals. These signals as such are not suited to the type of *symbolic reasoning* that the Inference Engine can perform. Chapter 8 therefore reviews methods to eliminate redundancies from signals and extract those features that are clinically meaningful, while simultaneously *validating* the signal so that no decisions will be based on erroneous data.

Chapter 9 then describes an expert system built with the SIMPLEXYS toolbox: a system which controls the infusion of the drug sodium nitroprusside in such a way that a patient's mean arterial blood pressure is stabilized at a specified lower than normal level.

Chapter 10, finally, will conclude that the SIMPLEXYS tools offer excellent support of all phases of an expert system's design. The SIMPLEXYS language formalizes the knowledge in a way that is well suited to step-by-step development of a knowledge base. The Rule Compiler detects and reports errors that may occur. The Inference Engine completes the expert system. And the Tracer/Debugger can find, analyze and report any errors that may still occur during the expert system's operation.

2. An introduction to expert systems

The body of literature about expert systems is tremendous, and it is impossible to summarize it here without doing the subject injustice. Globally, the literature can be divided into two categories: scientific books and articles on basic issues, methodologies and approaches; and more or less popular introductions and reviews of achievements, near-achievements and promises in this field. Expert systems are a fashion, and everybody seems to want to know about them.

The scientific literature clearly demonstrates, that there is no expert systems (or AI) *science* yet to speak of. If there *is* going to be such a science, we are currently in its very early stages, where 'the approach seems promising', but nobody is clear about what *the* approach should be. A great many different approaches are proposed, many are investigated on some small scale, only a small number on a larger scale, and only a very few have shown some success in practical applications. The problems seem more immense than expected.

This chapter starts with some expert systems basics: a brief historical review, some details on how different approaches represent and organize knowledge and on how that knowledge can be manipulated in expert systems. Section 2 then concentrates on the use of expert systems in real time tasks. In section 3 we consider one real time expert system, Fagan's VM, a major source of inspiration, in more detail. We conclude this chapter with demonstrating some important relations between expert systems, formal logic, and computer science.

2.1. Introduction

2.1.1. History

Artificial Intelligence (AI), also called *Machine Intelligence* (MI), emerged in the 1950's as one of the branches of what was to become known as computer science [Waterman, 1986]. The objective of AI research was to develop computer programs that in some sense could think. According to Minsky, 'Artificial Intelligence is the science of making machines do things that require intelligence if done by men'; and according to Feigenbaum 'AI research is that part of computer science that investigates symbolic, non-algorithmic reasoning processes and the representation of symbolic knowledge for use in machine intelligence'.

In the 1960's, investigations focussed on finding general methods for solving large classes of problems through attempts to simulate the process of thinking. A few 'General Problem Solver' programs appeared. This approach produced no breakthroughs; the more classes of problems a single program could solve, the more poorly it seemed to do on individual problems. So the emphasis shifted from general purpose *programs* to general purpose *methods* and *techniques*.

During the 1970's, AI research therefore concentrated on techniques like knowledge representation (encoding the problem so that a computer could easily solve it) and search

(controlling the search for solutions in such a way that it would not take too long or use too much of the computer's memory capacity). This strategy produced no breakthroughs either.

In the late 1970's, an important conclusion evolved: the problem solving power of a program comes from the *knowledge* it contains, *not* from the formalisms and inference mechanisms it employs. This conceptual breakthrough led to the development of special purpose computer programs, systems that were expert in some narrow problem area: *expert systems*. This new field of research, probably the most successful branch of AI research, has become very popular recently, partly, undoubtedly, through the efforts of the many AI experts who made their knowledge and experience accessible in books like 'Artificial Intelligence' [Winston, 1977], 'The Handbook of Artificial Intelligence' [Barr and Feigenbaum, 1981], both of which compile the major developments of the field's first 25 years, and the more specific 'Building Expert Systems' [Hayes-Roth et al, 1983]. A more recent overview of the field, as well as a collection of excellent papers, can be found in Gupta and Prasad [1988].

Currently, in the late 1980's, expert systems have become familiar tools to solve complex problems, but through their use a new problem surfaced: how to acquire the knowledge that is needed to solve such a complex problem. Thus new directions for research became on the one hand the development of machine learning techniques, through which (part of the) knowledge acquisition can be automated by having the machine discover general patterns underlying the data, and on the other hand the development of systems that do not depend on explicit knowledge but learn through experience or training (neural nets).

2.1.2. Organizing knowledge

When the term *knowledge* is used, it refers to the information a computer program must have before it can behave intelligently. This information can take the form of facts and rules, e.g. [Waterman, 1986]:

FACT: The spill material is sulfuric acid.
RULE: If the sulfate ion test is positive then the spill material is sulfuric acid.

A rule is often called a *production rule*. It produces a conclusion or some real-world action, such as an alarm, a suggestion or a comment. Facts[1] and conclusions of rules[2] need not be either true or false; it is possible to include a degree of uncertainty about the validity of a fact or the accuracy or applicability of a rule (see Bonissone and Tong [1985] for an overview of methods to do this).

The fact and the rule above belong together; the rule is a test procedure to establish whether the fact is true or not. In most expert systems, the truth of a fact can be established by more than one rule. In SIMPLEXYS, facts and rules are coupled; each fact is connected

[1] In SIMPLEXYS, *facts* (FACT rules) have a stricter interpretation.
[2] In SIMPLEXYS, rules will be shown to have a different form.

to one and only one rule that is dedicated to establish its truth, whether by some direct test procedure[1] or by evaluation of an expression which references other rules.

Many of the rules in expert systems are *heuristics* [Lenat, 1982]: rules of thumb or simplifications that more or less effectively limit the search for solutions. Heuristics are often needed if the task is difficult or poorly understood and defies rigorous mathematical analysis or algorithmic solutions. An algorithmic method guarantees to produce a correct or even the best solution to a problem, a heuristic method produces an acceptable solution most of the time, even if 'best' cannot be defined.

The term 'heuristics' is used in different ways. If a problem is mathematically tractable but very time-consuming (e.g. graph coloring), heuristics are considered additional, domain-specific pieces of knowledge that effectively confine the search space without in any way limiting the validity of the final conclusions. Most expert system problems cannot be not well defined in a mathematical sense, however. In these cases (e.g. chess), heuristics are usually considered to be clever short-cuts, that hopefully limit the search in such a way, that obviously unsuccessful searches are avoided. The cleverness and obviousness, however, are not full-proof and may even exclude the best solutions, although generally they lead to acceptable and occasionally even good solutions.

The knowledge in an expert system is organized in a way that separates the knowledge about the problem domain from the system's other knowledge, such as general knowledge about how to solve problems, how to acquire data or how to interact with a user. The collection of domain knowledge is called the *knowledge base* while the general problem solving knowledge is called the *inference engine*. A program with its knowledge organized this way is called a *knowledge based system*. This separation of knowledge makes it easier to design general procedures for manipulating the knowledge without being concerned with the specific features of the domain knowledge, and also, to design the knowledge base without being concerned with the specifics of how the knowledge is manipulated. How the system uses its knowledge is of the utmost importance, because an expert system must have both the appropriate knowledge and the means to effectively utilize this knowledge to be considered skilled at some task.

The inference engine handles the knowledge, the rules and the facts, as data. There is no simple, general way to characterize an inference engine. How it should be structured depends both on the nature of the problem domain and the way the knowledge is represented and organized in the expert system. Many high level expert system building languages, e.g. EMYCIN [van Melle et al, 1981] and SIMPLEXYS, have the inference engine built in as part of the language. Prolog contains a simple inference engine, but is also a convenient tool to build more specific ones. Lower level languages like LISP require the expert system builder to design and implement the inference engine.

---

[1] In what are called 'primitive' rules'.

All these approaches have advantages and disadvantages. A high level language with the inference engine built in means less work for the expert system builder, but it may also severely limit his design options.

2.1.3. Representing knowledge

There is a more or less standard set of knowledge representation techniques, any of which can be used alone or in conjunction with others to build expert systems. Each technique provides the program with some benefits, such as making it more efficient, more easily understood, or more easily modified. An excellent summary of the most important techniques can be found in Winston [1977] and in Barr and Feigenbaum [1981].

The four techniques most widely used are rules (by far the most popular), semantic nets, frames and blackboards [Hayes-Roth, 1985]; for an overview see Rich [1983], Niwa et al [1984], Milne [1988]. For the moment we will consider knowledge representation in the form of rules only.

Rule based knowledge representation centers on the use of statements with the following format:

IF condition THEN conclusion/action

When the current problem situation satisfies or matches the IF part of a rule, the THEN part of the rule is executed ('fired'). This execution may cause a match of another rule's IF part and so on; this matching action can produce *inference chains*, leading to additional knowledge or real world actions. Using rules this way is called *forward chaining* or *data driven* inferencing: each rule can activate one or more successors, which may cause an avalanche of actions (some of which may be conflicting).

Suppose, however, that instead we want to know whether a certain action needs to be performed or a certain conclusion can be reached. Forward chaining will certainly cause the action, but it can also cause numerous other actions; generally it will pervade the whole knowledge base. *Backward chaining* or *goal driven* inferencing avoids this. We start with the THEN part and check whether the IF part is satisfied. If not, we try to satisfy it by looking up those rules that have a THEN part corresponding with the IF part to be satisfied, continuing the search until we have our answer.

Rules can follow the rigorous mathematics of first order predicate logic, in the form of propositions written as *well-formed formulas*, which makes it possible to use the well understood mechanisms of propositional logic to process the rules and derive new facts. The programming language Prolog uses this approach[1].

---

[1] Without even mentioning the name Prolog, its 'spirit' is excellently described in Meltzer [1973].

Alternately, rules may describe conclusions to be derived and/or actions to be performed under certain conditions. These *production rules* will be part of a production system, which consist of:

- One or more databases that contain the information necessary for the task. Some parts of the data base may be permanent, other parts may contain temporaries necessary to solve the current problem.
- A set of rules, each consisting of a left side (a pattern) that determines the applicability of the rule, and a right side that describes the action to be performed if the rule is applied.
- A control strategy that specifies the order in which the rules will be evaluated and a way to resolve possible conflicts if more than one rule applies.

Production systems were first proposed by Post [1943] as a general computational mechanism. A production system has three basic components: a data base, a set of rules, and a rule interpreter; the invocation of rules is a sequence of actions chained by *modus ponens* (deduction). The reason for their popularity in AI is well stated by Newell and Simon [1972]:

> 'We confess to a strong premonition that the actual organization of human programs [i.e. in the human mind] closely resembles the production system organization.'

and

> 'In summary, we do not think a conclusive case can be made yet for production systems as *the* appropriate form of [human] program organization. Many of the arguments ... raise difficulties. Nevertheless, our judgment stands that we should choose production systems as the preferred language for expressing programs and program organization.'

Some rule based systems view rules in an even broader sense; they do not only *reason* but may also view the test *condition* as a *test procedure*, that can perform some unrevokable real-world action to reach a conclusion, such as to perform an extra measurement. In the example:

RULE: If the sulfate ion test is positive then the spill material is sulfuric acid

the condition 'the sulfate ion test is positive' might look in the data base to see if this fact is available; if not, it might order an automatic measurement; if this is not possible either, it could pose a question to the operator. An advantage of this strategy is, that the number of rules could be significantly lower than in a normal rule based system. However, testing for a condition in such a system creates side effects, and the order of rule evaluations may become very important.

Frame based and semantic net knowledge representations use a network of nodes connected by relations and organized into a hierarchy [Rich, 1983]. In a frame based representation, each node represents a concept that may be described by attributes and values associated with the node, such as default values, normal ranges and procedures that tell how an item that is needed but not present, can be found, acquired or evaluated. Thus side effects are carefully contained. Nodes low in the hierarchy automatically inherit properties of higher level nodes. This method provides a natural, efficient way to categorize and structure a taxonomy.

There is no agreement on the 'best' method for knowledge representation; for a discussion of the problems in and current thoughts about knowledge representation see Brachman and Levesque [1985]. In some applications rules may be superior, in others frames appear better. Much seems to depend on the earlier experience or the taste of the designer. It is not uncommon for comparable, almost equivalent, commercial systems to use very different ways for knowledge representation, although a convergence can be observed [Richer, 1986]. Indeed, for one and the same application, the representation may be changed drastically in order to achieve better performance [Kary and Juell, 1986; Neapolitan, 1988].

Nevertheless, several notions, however implemented, have become very important. *Chunking* [Newell and Rosenbloom, 1981] is the decomposition of knowledge into smaller elements, *chunks*, that are easy to comprehend and manipulate. *Goal-oriented* or *goal-structured* problem solving [Ernst and Newell, 1969; Newell and Simon, 1972] and its introduction of *goal hierarchies* [Rosenbloom, 1983] uses a type of chunking to decompose the task into a hierarchy of tasks that are successively easier to solve. For example, in an AND-hierarchy, a goal is successful only if all of its subgoals are successful [Rosenbloom and Newell, 1986]; in such a hierarchy, a *terminal* goal is a goal which can be fulfilled directly, without the need for further decomposition. All these notions are directly implemented in SIMPLEXYS.

2.1.3.1. Certainty factors

There is even more controversy in the Artificial Intelligence community about how to deal with uncertainty. The wide range of current views on this subject is compiled by Kanal and Lemmer [1986]. The best numerical model for uncertainty is probably an interval based calculus. There remain strong opinions, however, not only about how to interpret certainty and probability but also about its implementation into a numeric method.

Frequently, especially if the problem domain is large, probability factors are used to cut off those branches of the search tree that are not promising because they have a probability below a certain threshold. This is fine for (off-line) diagnosis, but in e.g. a context of patient monitoring it is very difficult to state how to treat an acute heart failure alarm that has a probability of 10%. Suppress the alarm? Leave the decision to the doctor? If the former, 10% of the acute heart failures are missed. If the latter, what is the use of certainty factors?

A more fundamental problem arises if we view (un-)certainty in terms of probability distributions. The propagation of probabilities is very much dependent on the correlations between the quantities used in the calculations (Bayes' law). These correlations are very difficult to estimate in a practical situation. Disregarding correlations may lead to meaningless results [Russek et al, 1983].

Feinstein [1973a, 1973b, 1974] exposes some of the reasons why probabilistic and statistical approaches have proven to be disappointing in the development of computer assisted clinical decision making procedures. Practical difficulties are:

- there is no foreseeable way to systematically evaluate the values to be assigned to alternative outcomes;
- the variables of concern are rarely independent;
- the alternatives are rarely comprehensively exposed;
- for meaningful application in most medical problems of interest, the total population must be fractionated into such numerous subgroups with which to match the individual patient that the resulting small numbers in each subgroup provide no valid basis for statistical decisions [Ward Edwards: 'the planet may not support a population large enough for the development of valid Bayesian inference schemes for most problems of interest in the real world'];
- problems of computability: the 'combinatorial explosion';
- probability distributions are unknown, and physicians do not use them in their practice;
- hypotheses are neither exhaustive nor mutually exclusive [Szolovits and Pauker, 1978];
- even skilled physicians disagree on findings;
- humans are poor in handling (small differences in) probability distributions;
- decisions are strongly affected by individual training and experience;
- diseases are multi-causal;
- diseases change over time, spontaneously or through treatment;
- there is a lack of universal definitions;
- definitions of 'disease' and diseases change over time;
- diseases are badly defined, have variable symptoms, are qualitative only, have a badly understood aetiology, and there are large individual and inter-group differences.

As Feinstein [1977] has pointed out, it is ironic that those who advocate allegedly objective probabilistic approaches to avoid the subjectivity of human mental function do not hesitate to provide subjective 'estimates' of *a priori* figures in their use of Bayesian inference or in the subjective estimation of decision thresholds. Furthermore, there is a growing acknowledgement that most classical probability approaches in medicine would more properly be classified as 'subjective probability'. These considerations have, although others do not agree with them, led us to shy away from certainty factors and the like.

2.1.3.2. Reasoning strategies

However the knowledge is represented, the expert system must 'reason' with it. Humans seem to employ different reasoning strategies and usually seem to be able to know when to use which. If we have a rule of the format

if P then Q

deduction tells, that Q is true if we know that P is true;
induction tells, that if P is true in a great many cases in which Q is true as well, that we can expect Q to be true when sometime in the future we observe P to be true;
abduction tells, that if Q is true there is some support to assume that P may be true as well.

Thus a rule may be used in different ways and have different meanings.

Deduction[1] is the standard mechanism in expert systems, because it is infallible. Some expert systems 'learn' (use induction) by keeping track of the correlation between observed occurrences of conditions and consequents. Poor results are obtained if the range of examples, that the system uses in learning, is not wide enough; if the system has observed twenty birds, all of which can fly, it may henceforth be quite certain that *all* birds can fly. A learning system may be able to use abduction too. Because induction and abduction are not infallible, they are usually combined with some type of certainty measure.

Many of the problems in the AI domain are too difficult to solve by direct means. Some type of 'reasoning' is necessary. This reasoning is usually nothing but searching: finding those rules that apply under the given condition, applying those rules, and again finding the rules that apply under the new conditions, and so on, until a solution (or all solutions) is obtained. Searching is a time-consuming operation, and for large applications it is essential to have 'smart' searching algorithms. For a good compilation of these methods see e.g. Nilsson [1971] or Rich [1983].

2.1.4. A critique of expert systems

The current approach to expert system technology has its critics, some of them quite vociferous. The name of the research area, 'expert systems', could by itself generate unrealistic expectations [Bobrow et al, 1986]; another name, 'knowledge-based systems' or 'knowledge systems' would better focus attention to the knowledge the systems carry, rather than imply expert behavior.

Warning against 'fully autonomous war machines that will respond to a crisis without human intervention', Dreyfus and Dreyfus [1986] state that 'after 25 years of research, AI

---

[1] Deduction takes a probabilistic form if the data have a statistical nature (see section 2.1.3.1).

has failed to live up to its promise, and there is no evidence that it ever will'. Some types of knowledge, e.g. bicycle riding, are commonplace, but this type of know-how is not accessible in the form of facts and rules. Intuition and understanding have been called 'holistic' or 'holographic'; detection of similarities with earlier experiences may be a better model than application of stereotype rules. 'In every area of expertise the story is the same: the computer can do better than the beginner and can even exhibit useful competence, but it cannot rival the very experts whose facts and supposed rules it is processing with incredible speed and accuracy'. Technically, there is no real distinction between expert systems programming and the normal activity of a programmer-analyst or a software engineer [Hart, 1986]; they just use different computer languages, and some of the implementations are based on outdated techniques.

We tend to agree with most of these criticisms. Much of the 'new' approach is not new at all, and frequently badly implemented. The almost ubiquitous use of the inherently interactive LISP as the implementation language makes, for large applications, searching the 'list' and skipping unwanted items a very time-consuming operation, as in computer language interpreters. Just as compilers were designed to combine ease of programming with efficient execution, rule compilers will be necessary to make expert systems more efficient[1]. Most current expert systems are, in the spirit of LISP (well described in Hofstadter [1986]), designed to be interactive, allowing easy replacement, editing and addition of rules and facts, but with a disastrous effect on efficiency. Real time applicability is now virtually non-existent and a convenient and efficient interaction with the real-world is frequently forgotten in current implementations.

Though all this may be true, even the critics admit that some of the best current expert systems 'produce performance often evaluated as being about 75 to 85 percent as good as experts' [Dreyfus and Dreyfus, 1986]. In fact, the various evaluations of the performance of MYCIN [Shortliffe, 1976] all suggested that it is as good as or better than most skilled human experts.

This possible superiority of expert systems is based on at least four factors. Harmon and King [1985] explain MYCIN's performance:

1. MYCIN's knowledge base, derived from some of the best human practitioners, is extremely detailed and is as comprehensive as that of most physicians in the domain of meningitis.
2. MYCIN does not overlook anything or forget any details. It considers every possibility. There is a popular saying amongst doctors that 'one has to think of the disease in order to recognize its symptoms'. MYCIN considers every disease it knows about.
3. The program never jumps to conclusions or fails to ask for key pieces of information. No matter how obvious the disease is, MYCIN methodically checks for all of the details and considers all alternatives.

---

[1] 'It is possible to use techniques such as decision trees and discrimination networks to "compile" the rule base and speed the search. ... Most developers of expert systems have no idea what these techniques can do for them' [Milne, 1988].

4. MYCIN is maintained at a major medical center and is, consequently, completely current. Several of its therapy recommendations are based on recent data published in specialized journals. Such information is not in textbooks, and would be known only by specialists who monitor the journals and who remember to incorporate new information into their diagnostic procedure.

These factors indicate some human frailties, that are absent in well-designed, well-maintained expert systems.

2.2. Real time expert systems

When the rate of information flow is too great, humans have a tendency to overlook relevant information, to respond inconsistently, to respond too slowly, and to panic. 'The principal reason for using real time expert systems is to reduce the cognitive load on users or to enable them to increase their productivity without the cognitive load on them increasing' [Turner, 1986]. Much of this cognitive load has to do with the pressure of time.

An excellent overview of the current state of the art in real time expert systems can be found in Laffey et al [1988], where the problem is introduced as follows:

> 'A knowledge-based system operating in a real time situation (for example, crisis intervention or threat recognition) will typically need to respond to a changing task environment involving an asynchronous flow of events and dynamically changing requirements with limitations on time, hardware, and other resources. A flexible software architecture is required to provide the necessary reasoning on rapidly changing data within strict time requirements while it accommodates temporal reasoning, nonmonotonicity, interrupt handling, and methods for handling noisy input data.'
>
> 'The complexity of these systems is increasing rapidly along three dimensions: (1) the number of functions controlled, (2) the rate at which the functions must be controlled, and (3) the number of factors that must be considered before a decision can be made.'

The quantity of literature on real time expert systems is not overwhelming. A recent set of papers dedicated to AI in real time control can be found in Rodd and Suski [1989].

2.2.1. Definitions of real time

O'Reilly and Cromarty [1985] mention several interpretations of what is meant by *real time*. The most common usage is 'fast': a real time system is a system that processes data quickly. Also: 'perceptually fast', or 'faster than a human can do it'. A better definition is: 'fast enough', or 'the system responds to incoming data at a rate as fast or faster than it is arriving'.

Two more formal definitions are:

A system exhibits real time behavior if it is 'predictably fast enough for use by the process being serviced' [Marsh and Greenwood, 1986].

A system exhibits real time behavior if 'there is a strict time limit by which the system must have produced a response, regardless of the algorithm employed' [O'Reilly and Cromarty, 1985].

Simmonds [1989] views a real time system as an intermediate between a human and a machine, an interface, a means of control of an otherwise uncontrollable plant:

For on-line computer based systems 'real time' means working at the tempo of people on one side and working at the tempo of the plant on the other side.

Laffey et al [1988] stress the importance of a guaranteed response time:

'Response time is the time the computer takes to recognize and respond to an external event. This measure is the most important in real time applications; if events are not handled in a timely fashion, the process can literally go out of control. Thus the feature that defines a real time control system is the system's ability to guarantee a response after a fixed time has elapsed, where the fixed time is provided as part of the problem statement. If, given an arbitrary input (or event) and an arbitrary state of the system, the system always produces a response by the time it is needed, then the system is said to be real time'.

2.2.2. Real time expert systems are different

In a real time system, the data to be analyzed are not static. Incoming sensor data are not durable or have a decay in validity with time or cease to be valid because events have changed the state of the system.

Two approaches to the processing of real time data are possible. In the first approach, a complete new set of data is acquired and processed whenever an 'event' occurs; an 'event' is a (significant) change of one or more of the input data. This is the familiar expert systems approach: each data set resembles a filled out form that has to be analyzed, and an event is just a message that a new form has arrived for analysis. This new form probably has most of its data in common with the previous one, and consequently most of the data processing time may be 'wasted'. If the analysis is fast enough, that will not constitute a problem, however. SIMPLEXYS uses this approach.

The second approach does not 'waste' time; it does not reevaluate conclusions based on unchanged data. In this second approach, an 'event' or change of data just retracts those

conclusions that were based on the now outdated data and reevaluates only the rules that provided those conclusions, but now employing the new data. However, the nonmonotonicity of the incoming data (and thus of the conclusions deduced) calls for *truth maintenance*, a process that at all times keeps the conclusions consistent with the data. Truth maintenance is a complex and time consuming process, and therefore this approach may not be more efficient than the former. Moreover, truth maintenance and retraction of earlier conclusions is possible only, if the system is purely logical, *not* if conclusions can have 'side effects' such as firing a missile or applying a drug. In some systems, the 'garbage collection' that is necessary, once in a while, to discard old, irrelevant data might also interfere with timeliness.

With either approach, partial or complete failure of parts of the system, especially temporary failure of one or more of the sensors, should not necessarily imply that the system stops functioning; the system should be designed for 'graceful degradation'. Missing data will be common, and the validity of some of the data can decay with time, e.g. due to a degradation in sensor performance.

Events may be either synchronous or asynchronous. If the events occur synchronously with some clock, the time *available* for analysis is constant. However, the time *necessary* for an analysis of the data may depend on the data. If the time necessary is *on average* longer than the time available, the load is of course too heavy for the system. If the time necessary is *only sometimes* longer than the time available, buffering of the data may be a solution, if this does not degrade response times too much. In case of a system overload, another option may be to decrease the rate of the analysis.

Asynchronous events are events that fail to conform to any *a priori* known schedule; the time available for analysis now depends on the time between events, which may be completely unpredictable. In some processes there may now be a large probability that some events will succeed each other so fast that the processing time of some events will be too short. Temporarily buffering new data may again be a solution, response times allowing (the response time will now be random as well). Since events can vary in importance, it is also an option not to process an event as long as more important ones demand processing time. Many extra options exist, besides buffering data, if a (temporary) system overload forces a 'focussing of the attention', such as adjusting the set of sensors employed (process less information) or selecting different knowledge sources (choose 'specialists' that process the data more globally). The manner in which the attention is focussed may be chosen to depend on the circumstances, as well.

Another popular approach to guaranteed response times (e.g. in chess computers) is to store, at all times, the best[1] response so far, and to provide this 'best' response when the deadline expires. Such a simple approach is not adequate if many different types of event can occur.

---

[1] It is, however, often very difficult to find a criterium for what must be considered 'best'.

An important feature of real time systems is an integration with procedural components (conventional real time software) for data acquisition and compression, signal processing, feature extraction, application-specific input/output, etc.

Real time systems cannot do without some form of temporal reasoning. Time is an important variable, and reasoning about past, present and future may all be equally important. Reasoning about sequences of events may be necessary as well.

2.2.3. Problems in real time expert systems research

Laffey et al [1988] review several real time expert systems; they mention applications in aerospace, communications, medicine, process control, and robotics. They also give an overview of real time expert system tools. They consider some of the most important current theoretical issues to be:

- the slow execution speed of rule-based systems (90% of the time is spent in searching);
- a production system that uses forward or backward chaining takes exponential time [O'Reilly and Cromarty, 1985]; worst case analysis probably results in unacceptable response times;
- frame languages using only simple inheritance (each class has at most one super-class; only 'is-a' links and instance) are suitable (searching is then limited to linear lists); most frame languages are more complex, however, resulting in exponential searching times; 'no one seems to be considering the ramifications of using exponential-time algorithms in a real time application';
- the use of more processors in parallel does not help much (a factor 10 at most), depending on programming style;
- guaranteed response times are difficult to achieve; O'Reilly and Cromarty [1985] discuss the deficiencies of the hand-tuning of response times:
  - there are no general methods; ad hoc methods need to be reinvented for each new problem;
  - performance becomes brittle on changes in problem specification and type or quantity of data;
  - tuning is very time-consuming (it is a difficult unconstrained search problem);
  - there is no formal basis, there are no guarantees.

They state as the currently most pressing problems to be solved:

- the shells are not fast enough;
- the shells have little or no capacity for temporal reasoning;
- it is difficult to integrate efficiently with conventional software;
- there are little or no facilities to focus attention;
- there is no integration with a real time clock;
- there are no facilities for handling asynchronous inputs;

- there is no way of handling software-hardware interrupts;
- systems cannot efficiently obtain input from external non-human stimuli;
- there are no methods to verify and validate the shell or the knowledge base;
- the shells cannot guarantee response times;
- shells run on hardware not built for harsh environments.

And the most pressing requirements are stated as:

- an efficient integration of numeric with symbolic computing;
- continuous operation (garbage collection is now a problem);
- a focus-of-attention mechanism;
- an interrupt-handling facility;
- optimal environment utilization; i.e. compiled instead of interpreted code;
- predictability (garbage collection periodically comes into action, but at unpredictable times and for unpredictable periods of time);
- temporal reasoning facilities; maintaining, accessing, and evaluating historical data must be possible;
- a truth maintenance facility; data quality decays in time.

2.2.4. A critique of real time expert systems

Despite Laffey et al [1988]'s observation that 'considerable effort is being put into developing real time systems, a much more difficult area than traditionally has been approached using expert systems', they also reach the following conclusions:

- Expert system developers have often tried to apply traditional tools to applications for which they are not well suited. Tools specifically built for real time monitoring and control applications need to be built. An immediate goal should be the development of high-performance inference engines that can guarantee response times.
- Trying to apply current shells to real time domains is like trying to use Prolog for a number-crunching application or Fortran for a symbolic processing application.
- 'Real time expert systems are real hard to develop' (Robert C. McArthur of the Arthur D. Little Company, quoted in Marsh [1986]).

Our own reviews of available products[1] at the time SIMPLEXYS was started confirmed the above. The SIMPLEXYS experiment was started to specifically address the issues mentioned above.

---

[1] Such a review does not easily lead to well-defined conclusions. Product documentations usually did not mention aspects that were important to us, such as worst case performance. Because of product complexity, it takes too much time to exhaustive test more than a few shells. Product pricing often even prevents the latter.

2.3. VM: a real time expert system

VM (Ventilator Manager) provides diagnostic and therapeutic suggestions about the ventilation (artificial respiration) of postsurgical patients in an intensive care unit (ICU). The system identifies possible alarm conditions, recognizes spurious data, characterizes the patient's state, and suggests useful therapies. The system interprets quantitative measurements from an ICU monitoring system, such as heart rate, blood pressure and data regarding a mechanical ventilator that provides the patient with breathing assistance, by applying knowledge about patient history and expectations about the range of monitored measurements. VM is a rule based system implemented in INTERLISP. It was developed at Stanford University and tested at the Pacific Medical Center in San Francisco and at the Stanford University Medical Center.

The discussion in this section is based on the literature about VM [Fagan, 1978; Fagan et al, 1979; Fagan, 1980], from which we select those aspects which are relevant in SIMPLEXYS.

2.3.1. An introduction to VM

In many aspects VM resembles the type of system that we specifically address in this study. Moreover, it is a real time system, and few of those have been exhaustively described in the open literature. The real time behavior of the system is indicated by the fact, that processing 24 hours of patient data takes about 15 minutes of central processor computation time, allowing one processor to monitor all patients in the ICU. This speed was attained despite the fact, that VM was written in LISP: it is a small system with about 50 rules[1].

VM assists the clinician by providing diagnostic and therapeutic suggestions for the control of the equipment that provides mechanical breathing assistance. Its setting is an intensive care unit where patients in the period immediately following cardiac surgery are cared for and weaned off the ventilator. Its suggestions are based on the interpretation of some 30 variables of the patient's status. Most measurements (heart rate[2]; pulse rate[3]; systolic, diastolic and mean arterial blood pressures; inspired, expired and minute respiratory volumes; respiratory rate; mean and maximum airway pressures; end-expired pCO_2, inspired pO_2; oxygen uptake and CO_2 production; thoracic plus lung compliance; patient-respirator fighting; in- and expiration times) are collected every two minutes (fast mode) or every ten minutes (normal mode); some measurements (temperature, blood gases) have to be entered manually. Otherwise there is no user interaction; the system is fully autonomous.

---

[1] We assume that most data were analyzed at 10 minute intervals (see below). A quick calculation shows that at 5 second intervals the processor would not be able to keep up with the data.

[2] Derived from the electrocardiogram.

[3] Derived from the invasively measured arterial pressure.

The *AI research goal* was to extend existing knowledge representation and manipulation techniques for a dynamically changing medical environment, in which the interpretation process is repetitive, in order to capture the on-going process. The *medical goal* was to assist the ICU clinician by adding a measurement interpretation and evaluation capability to the existing monitoring system. VM addresses these patient monitoring problems:

- the current inability to form a concise presentation of all the important measurements;
- the need for interpretation of measurement values with respect to historical information about changes in patient status and therapy;
- the lack of ability to directly relate measurement values with therapeutic recommendations.

It was designed to perform these tasks:

- detect possible measurement errors;
- recognize untoward events in the patient/machine system and suggest corrective action;
- summarize the patient's physiological status;
- suggest adjustments to therapy based on the patient's status over time and long-term therapeutic goals;
- maintain a set of patient-specific expectations and goals for future evaluation by the program.

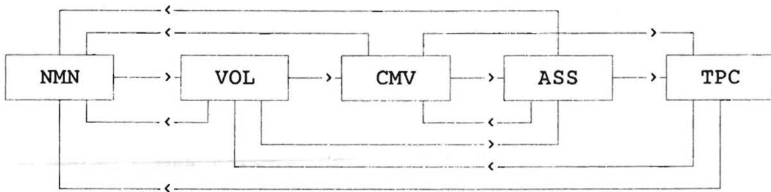


Figure 2.1. The VM monitoring context.

VM operates approximately as follows: every 2 or 10 minutes a set of patient measurements is collected from the monitoring equipment and analyzed. The analysis can lead to alarms and/or suggestions about the therapy. But the system had no way to directly measure whether the patient is being artificially respirated or not; that has to be concluded from the signals as well. Worse: such a conclusion often can only be derived from several sequential sets of signals, e.g. because establishment of the 'stability' of the respiratory rate takes some time[1]. But such 'context' information (i.e. is the patient being artificially respirated or not) is essential for the way in which the measurements should be interpreted. It is therefore crucial to store such information.

[1] If a patient is respirated, his respiratory rate is dictated by the equipment and hence very regular, unlike in spontaneous breathing.

Figure 2.1 shows VM's monitoring 'context'. The patient can be either:

- not monitored (NMN); no measurements are available;
- on volume ventilation (VOL); the patient is being ventilated, but it takes some time before the system can establish from the regularity of the respiration whether ventilation is controlled (CMV) or assisted (ASS); thus VOL is actually hierarchically at a higher level than CMV and ASS;
- on controlled mandatory ventilation (CMV); the ventilation is fully controlled by the respirator;
- on assisted ventilation (ASS); the patient is allowed to breathe spontaneously, but the respirator supports the breathing;
- on T-piece (TPC); the respirator has been disconnected but some measurements are still available.

When certain conditions arise, a context switch is necessary. So-called transition rules determine whether such a context switch is necessary. The transition from NMN to VOL, for instance, depends on the presence and physiological validity and stability of some respiratory signals during some time (this transition takes some time because the patient must be connected to the respirator; during this time the signals may be quite erratic).

The patient (or rather: the monitoring process) can be in one of five states, NMN through TPC. The links between these states describe what in SIMPLEXYS terminology is called the *protocol*. Normally, the protocol is for the patient to go from NMN to VOL to CMV to ASS to TPC to NMN, but if e.g. extubation is too early, the patient may go back to VOL and restart from there. Other, less normal, transitions may occur as well.

2.3.2. Knowledge acquisition

Discussions with the medical expert, dr. John Osborn, about patient records (24-hour plots of patient data) showed that the clinician:

- compares the values of the measurements against typical values, noting abnormally high or low readings;
- looks for significant events: places where measurements are particularly unstable or changing rapidly;
- seeks corroborative evidence for an interpretation by closely aligning several of the traces and verifying that an appropriate pattern also shows up in related readings;
- notes the slopes of various computed indices that demonstrate the status of the patient.

There were several things to note about the expert's reasoning activities. Much of this activity assumes that there exist standard ranges, trends and stereotypical events that can be used as a guideline for recognizing problems. Individual data points that are far removed from the expected trends are for the most part ignored. Potential events suggested by a

trend in one measurement are compared with other measurements that are either typically correlated with the original finding or would be correlated if the hypothesized event had taken place. The former case is to determine if a change could be due to a malfunction of a sensor. An example of the latter case is verifying that a sudden rise or drop in the blood pressure happens concurrently with loss of the airway pressure measurement, implying that the patient has been taken off the ventilator. Here the choice of additional measurements is dependent on the hypothetical cause of the problem. In general, the activity is to peruse individual measurements looking for unusual situations and then to verify their existence by comparison with changes in other measurements.

In the ICU the interpretation of measurements has a different character than such a retrospective analysis by an expert. The clinician nor the expert system can 'look ahead' and must therefore be conservative and treat an abnormal reading as valid, at least temporarily.

The knowledge incorporated into VM is a symbolic model of the various therapeutic maneuvers for ventilator assisted patients. This model takes the form of a list of possible therapies linked together by a set of production rules that recognize and propose transitions between the various therapies. This model represents knowledge about ICU activities that could affect the patient's status as reflected in physiological measurements taken at the bedside. This model is used to 'customize' the data interpretation as the patient's situation changes over time. This context-sensitive interpretation is accomplished by the recognition of the therapeutic state and adjusting expectations or limits on each of the measured variables.

2.3.3. Data and the data base

The knowledge in VM is based on relationships between the various data such as respiration rate, sex of the patient and hyperventilation. Each of these data is associated with a value: the respiration rate is high, the sex of the patient is male and hyperventilation is present. The values associated with each datum may vary with time: hyperventilation was present for 30 minutes starting 2 hours ago. The data have been classified as:

```
constant        examples: surgery type, sex;
continuous      examples: heart rate, blood pressure;
volunteered     examples: temperature, blood gases;
deduced         examples: hyperventilation, hemodynamic status.
```

The following is a list of possible 'properties' of data:

```
DEFINITION:     text describing the datum;
USED-IN:        list of rules that use this datum;
CONCLUDED-IN:   list of rules where this datum is concluded;
EXPECTED-IN:    list of rules where expectations about this datum
                are made;
GOOD-FOR:       length of time that a measurement can be assumed to
                be valid; if missing, then it must be recomputed,
                acquired if possible, else assumed unknown;
UPDATED-AT:     last time the datum was acquired or concluded.
```

The properties USED-IN, CONCLUDED-IN and EXPECTED-IN are used to specify how the data are formed into a network of rules; they can be used to guide various search strategies. The properties UPDATED-AT and GOOD-FOR are used to determine the validity of the datum over time. The GOOD-FOR property can also be a pointer to a, possibly context-dependent, rule.

A large quantity of data is presented to the system compared with typical knowledge based systems. No attempt is made to retain all these data permanently in memory; only the most recent information (approximately one hour's worth) is utilized to form conclusions. However, the conclusions based on the original data, which are stored much more compactly, are maintained throughout the ICU stay. Thus the measurement values are replaced by symbolic abstractions over time.

2.3.4. Rules and the rule base

Knowledge is represented in the form of production rules. Rules have as uniform a structure as possible. The motivation behind this rule structuring was an easier rule acquisition; this structure makes the rules look like frames.

An example of a rule:

```
STATUS-RULE:   STABLE-HEMODYNAMICS
DEFINITION:    defines stable hemodynamics based on blood pressures
               and heart rate
APPLIES-TO:    patients on VOLUME, CMV, ASSIST, T-PIECE
COMMENT:       look at mean arterial pressure for changes in blood
               pressure; look at systolic pressure for maximum
               pressure
IF
   HEART RATE is ACCEPTABLE
   PULSE RATE does NOT CHANGE by 20 beats/minute in 15 minutes
   MEAN ARTERIAL PRESSURE is ACCEPTABLE
   MEAN ARTERIAL PRESSURE does NOT CHANGE by 15 torr in 15 minutes
   SYSTOLIC BLOOD PRESSURE is ACCEPTABLE
THEN
   HEMODYNAMICS are STABLE
```

The first line gives the rule type and its symbolic name. The definition and comment parts are for documentation purposes only. The applies-to part defines the therapeutic context in which the data are collected, which can influence their interpretation; it also limits searches. The if part holds relations, all of which must be true to lead to a conclusion; no certainty factors are used in the rule evaluation, though the mechanism is available.

Some of the relational components, e.g. ACCEPTABLE, may depend on the context. In that case, they are defined on entering this context. The THEN part may lead to a conclusion, to a suggestion to the clinician or to new expectations about the future values of

data, e.g. that some variable should be in some range at some point in the future. An abbreviated example of such an initialization rule:

```
INITIALIZING-RULE:     INITIALIZE-CMV
DEFINITION:            initialize expectations for patients on
                       controlled mandatory ventilation (CMV) therapy
APPLIES-TO:            all patients on CMV
IF ONE OF:
  PATIENT TRANSITIONED FROM VOLUME TO CMV
  PATIENT TRANSITIONED FROM ASSIST TO CMV
THEN EXPECT THE FOLLOWING:
```

		acceptable				
			ideal			
	very low	low	min	max	high	very high
MEAN PRESSURE	60	75	80	95	110	120
HEART RATE			60	110		
EXPIRED pCO2	22	28	30	35	42	50

The mean pressure will thus henceforth be considered 'ideal' if it has a value between 80 and 95 mmHg, 'acceptable' if it has a value between 75 and 110 mmHg, 'unacceptable' if its value is below 75 or above 110 mmHg, and 'very unacceptable' if its value is below 60 or above 120 mmHg. Note that for the heart rate not all expectation bounds are defined.

For the physiological data, the following types of expectation bounds were established:

```
ideal              the desired level or range of a measurement;
acceptable         beyond these values corrective action is needed;
unacceptable       corrective action is necessary;
very unacceptable  the data are extremely out of range, a severe
                   condition exists;
impossible         the data have no physiological origin.
```

These bounds are not static but adapt with the patient's situation or *context*, e.g. ventilator support type, through the execution of the appropriate initialization rules. There was no need to generate higher level expectations, such as 'hyperventilation expected in 30 minutes', although such an extension would be possible.

In addition to 11 status rules and 6 initialization rules, the system has three other types of rules: 13 instrumentation rules, 11 transition rules and 13 therapy rules, a total of 54 rules. An instrumentation rule examines the measurements to determine if they are consistent and within physiologically meaningful limits. A transition rule detects when a patient's state has changed, e.g. when the patient starts to breathe on the T-piece. This is necessary because there is no manual input of such changes and they cannot be measured directly. Therapy rules exist in two classes: long-term therapy assessment (e.g. when to put the patient on the

T-piece) and the determination of a response to a clinical problem, such as hyperventilation or hypertension.

2.3.5. Inferencing

The knowledge base includes rules to support five reasoning steps that are evaluated at the beginning of each new time segment:

1. characterize measured data as reasonable or spurious;
2. determine the therapeutic state of the patient (ventilation mode);
3. adjust expectations of future values of measured variables if the patient's state changes;
4. check physiological status;
5. check compliance with long-term therapeutic goals.

Each of the rule groups corresponding to the steps above is considered in order. Each rule is examined to determine if it applies to the current context. The premise of the rule is examined to determine validity and the appropriate conclusions are recorded, as well as expectations on the future ranges of measurement values. Suggestions to clinicians are also printed out.

The rule interpreter is based on the MYCIN interpreter with the following major changes: forward chaining instead of backward chaining; validity checking for previously acquired data (in MYCIN the data are manually entered and assumed error-free); cycling through the rules each time new information is available (MYCIN analyzes its data only once).

The first change was motivated by the application. It was deemed necessary because more than one conclusion can be reached at a time, and the small number of rules made it entirely feasible.

The second change reflects the clinical reality of loss of validity of old measurements. The reliability of the stored value is determined by evaluating either a time constant (for variables that change predictably over time) or a rule (for cases in which the assessment of a value's reliability is dependent on context specific information). If a measurement is concluded to be spurious or outdated it is treated as if it were unknown, requiring alternate methods for determining the status of the patient. The third change follows from the fact, that the application is real time.

Conclusions have an associated time or time range. Identical conclusions made at several adjoining rule set evaluations are considered continuations of the first conclusion and represented by a pair of times corresponding to the first and last such conclusion. A list of these intervals summarizes the history of a particular conclusion. The evaluation of a rule clause such as 'patient hyperventilating for greater than 20 minutes' is made by direct examination of the time intervals associated with each conclusion, rather than looking at the original measurements. Expectations are associated with the appropriate measurement and

are classified by duration and type, such as the upper limit of the acceptable range. Expectations can persist for a fixed interval such as 'for twenty minutes starting in ten minutes' or for the duration of one or more clinical situations, e.g. while the patient is on ASSIST.

The actual processing of a rule is carried out by a series of LISP functions that test conditions. Each function has a well defined semantic interpretation and provides the primitives for encoding the knowledge base. The translation between an external format 'RESPIRATION RATE > 30' and the internal format '(MCOMP RR > 30)' is made by a parser. The MCOMP function is given a parameter name (RR), a relational operator (less than, equal, greater than) and a number with which to compare. The execution of the MCOMP function returns a numerical representation of true, false or unknown, based on the current value of the respiration rate. SIMPLEXYS is based on the same three-valued logic.

2.3.6. A review of VM

It would not be fair to judge VM by the requirements of today, as set forth in section 2.2. In fact, VM was, as far as we could determine, the first expert system described in the open literature that systematically paid attention to the demands of a real time environment.

In the development of VM, no attention whatsoever was devoted to efficiency considerations [Fagan, personal communication, 1988]. There was never a demand for it; VM is a small system, and despite the fact that it was written in LISP, its cycle time was so slow (10 minutes, occasionally 2 minutes), that timing problems never occurred.

VM did pay attention to another important requirement, however; its goal was to investigate methods for reasoning with time. It introduced the notion of time-varying contexts, each of which could define different conclusions to be derived. It also introduced time information connected to conclusions, so that it would be possible to check *how long* a certain condition had been true. In these respects, SIMPLEXYS builds upon VM.

2.4. The relation between expert systems and formal logic

Brachman and Levesque [1985]:

> 'The role of formal logic in Knowledge Representation (KR) has been hotly contested from the beginning. ... Part of the problem has to do with the history and goals of Knowledge Representation and their differences from those of symbolic logic. After Leibniz, the next big push in formal logic was the work of Frege at the turn of the century who, along with Russell, Peano, and others, gave logic much of the flavor it has today. The goal of this early work was to put mathematics and mathematical reasoning on a sound theoretical footing. Indeed, until recently, the major application of symbolic logic

and one of the great successes of twentieth century mathematics, has been the analysis of formal theories of sets and numbers.
The goals of Knowledge Representation, however, even at the very beginning, were quite different, and perhaps much more in line with Leibniz' original dream. Knowledge Representation schemes were used to represent the semantic content of natural language concepts, as well as to represent psychologically plausible memory models. In neither case was there a clear relationship to formal *languages* of any kind. Gradually, however, Knowledge Representation schemes began to be used as a very flexible and modular way to represent the facts that a system needed to know to behave intelligently in a complex environment'.

Horvitz [1986] discriminates between two approaches to expert systems:

The *descriptive* approach. Descriptive models summarize complex behavior by *describing* phenomenology without resorting to fundamental axioms, which may not be available. They employ empirical, informal models and are most often rule-based. Rules of logical inference (e.g. modus ponens and unification) are used in automated deduction. An example is MYCIN [Shortliffe and Fagan, 1982]; originally MYCIN used an ad hoc calculus for evidence combination, that was internally inconsistent. With this approach, inconsistencies are often hard to avoid; if they exist, they lead to unpredictable behavior.

The *axiomatic* approach. This approach leads to expert systems that are necessarily *consistent* with desired properties, strive for a *normative* theory for reasoning, have a consistent *revision of belief* and control of *information acquisition*. Examples of axiomatic theories for belief revision are: probability [Pearl, 1985]; fuzzy logic [Gaines, 1978]; Dempster-Shafer theory [Shafer, 1976]; certainty factors [Shortliffe and Buchanan, 1975]; multi-valued logics [Gaines, 1978]. Examples of theories for controlling information acquisition are: information theory [Shore, 1986]; decision theory [Pratt et al, 1965; Raiffa, 1968]. These formal models are very helpful when a system is modified, since they allow a crisp prediction of changes in system behavior in response to system modifications. This approach also has its problems, however: the low level reasoning leads to an enormous thirst for computing power; explanations are also very difficult due to the low semantic content of the logical elements.

Some of the currently most important issues in the representation of knowledge are discussed in [Brachman and Levesque, 1985]. These include expressive adequacy and reasoning efficiency.

Expressive Adequacy. What are the standards for measuring the expressive adequacy of a representation language? Should every language be a notational variant of the language of full first order logic? At the very least, a knowledge representation language should be very clear about precisely what knowledge it is representing, something that has not always been true in the past.

Reasoning Efficiency. It is one thing to come up with a representation of knowledge that is sufficient to logically imply any fact of interest, but quite another matter to actually calculate these implications, especially when the representation language is expressive enough. How can we balance the need to provide correct, rational inferences with the need to produce them in a timely fashion?

First order logic is not necessarily best because 'the reasoning process in first order predicate logic is undecidable, and yet ... *in most cases* [our italics] a process of reasoning terminates after a reasonable length' [Mamdani and Efstathiou, 1988]. Both expressive adequacy and efficiency arguments may be the reason to invent more complex logics as a basis for inferencing [Mamdani and Efstathiou, 1988], even though they relinquish some mathematical exactness. The rule-based expert systems methodology is very flexible in the type of logic that it allows. Although the core is always an adaptation of the automated logical inference methodology called *production systems* [Simon, 1972; Davis et al, 1977], these adaptations range from very informal to extremely formal in a mathematical sense.

Many authors have suggested that some exotic logic, especially 'fuzzy logic', is necessary in order to capture the essentially imprecise nature of human deduction, but the usual arguments advanced for the use of fuzzy logic are hardly convincing: no logic, however fuzzy, can adequately capture 'human' logic. Some kind of formal logic is required to prescribe to the computer in full detail how its 'automated reasoning' should be performed; exactly *which* logic is most convenient may be dependent on the type of application. Brachman and Levesque [1985]: 'The emerging view is more that representation languages are perhaps non-standard logical languages, non-standard in their syntax, semantics, and use'. But logical nonetheless.

SIMPLEXYS combines the advantages of the descriptive approach (a high expressiveness, resulting in compact knowledge bases and efficient reasoning) with the advantages of the axiomatic approach (a formal logic, resulting in correct knowledge bases and correct inferencing).

2.5. The relation between expert systems and computer science

The notion of a well-structured knowledge base apart from the program that manipulates that knowledge is a good idea, that gears well with common computer engineering notions like *top-down programming, program modularity* and *information hiding.*

Top-down programming refers to the practice in which the program that is being designed is meaningful (performs useful work) at all intermediate design stages, even though details have not yet been implemented. This implies that the program can also be *tested* at every design stage. Unimplemented subprograms are replaced by a 'stub', that, during testing, just announces that the subprogram is 'working', even though it does not do anything yet. This programming practice allows one to concentrate on the main issues, disregarding

details for the moment; a detail then becomes a main issue at a later time. Programs are thus constructed through *iterative refinement.*

In knowledge base terms, top-down programming refers to the practice of starting with a small, comprehensible knowledge base that just has a few high-level concepts, without as yet worrying how those concepts will be implemented. This knowledge base can be tested immediately, both syntactically, semantically and in simulations. Each refinement step is then the introduction of new concepts, that 'explain' or 'define' higher level concepts. This design strategy helps in assuring and maintaining the correctness of the knowledge base right from the start of its design.

Program modularity refers to the practice of keeping each section of the program so small, that it can be comprehended without much effort. If a program section becomes too large, a sub-section is split off and becomes a section by itself. Each section will tend to incorporate just one or a few well-defined concepts and/or operations.

In knowledge base terms, the knowledge is split up in 'chunks', each of which implements one concept and possibly additional information about or operations on this concept.

Information hiding refers to the fact, that low level details become unimportant, and in fact ought to be hidden, at a higher level. A procedure *sort* might exist, that sorts an array; when it is used, we are not concerned with the details of *how* the sorting is done.

In knowledge base terms, high level concepts can be used as such, without concern about the details of *how* they are implemented and how they will be evaluated. That concern belongs to another level.

The knowledge based approach is particularly exciting in research applications where the knowledge to be incorporated into the system is still only partly known. One of the major problems in programming complex applications has always been the maintenance of the program as it evolves. By separating the knowledge, e.g. in the form of rules, from the program body, the update and maintenance of the program may be limited to the rule base or part of it only, allowing a much faster and easier generation of a sequence of prototype systems.

This does not mean, that in the final code the rule base as such needs to exist. Any program transformation operation that makes the program more efficient without altering its behavior could be used. It has frequently been observed that the order of the rules in the rule base has a great influence on the efficiency of the resulting system. It might be possible to order the rules in such a way, that the search efficiency is optimized, or, preferably, that searching is eliminated altogether (see also section 4.11). This is an example of a program transformation operation that could possibly be carried out mechanically and result in a more efficient, or even the most efficient program.

3. The machine monitor

This chapter discusses the machine problem solving methodology on which SIMPLEXYS is based. Computers must be prescribed, in full detail, which problem or problems must be solved, in which order, and how. This is the issue we explore in this chapter. We do not examine the most central problem of AI, the *frame problem*, which can be stated as: which are the relevant aspects of a problem situation or context, and how are the important features selected while ignoring the 'infinite' rest [Kelly, 1987].

Section 3.1 discusses problem solving in medicine and introduces the *hypothesize and test* strategy and the medical *protocol*; both are fundamental SIMPLEXYS notions. Section 3.2 describes some problems in machine reasoning. The machine must not only solve problems, it must solve them in an expert fashion (for some of the properties that make an expert an expert see Tullemans [1987]); some 'expert' problem solving features should exist in a machine expert. Section 3.3 describes the problem solving methodology on which the SIMPLEXYS programming language is based, as well as its knowledge taxonomy and how it is implemented. Finally, section 3.4 outlines some of the issues that played a role in the design of the SIMPLEXYS computer language.

3.1. Problem solving in medicine

Medical practice is an extraordinarily complex human activity. The required knowledge is extensive; ideally, a comprehensive pathophysiological model of human function and disfunction at the macro, intermediate and micro level should be available. But the human organism is extremely complex; only very partial knowledge is available. And any individual practitioner commands only a small part of that knowledge.

These gaps in the medical knowledge structure must be bridged by art; decisions must be made based on uncertainty and incomplete knowledge. The uncertainty is compounded by the numerous homeostatic mechanisms, which, even in extreme disorders, may act to mask failure of a 'component' of the system through some type of compensation, and by the extreme variations in individual response. Also, often it is unclear which data would be helpful, and the quality of the obtained data may be unknown as well. Many of the data that are required for decision making must be obtained from others, especially from the patient himself. The elicitation of the necessary data is an art in itself.

Medical reasoning [Feinstein, 1973a, 1973b, 1974, 1977] consists of several steps [Fink and Galen, 1982], although this dissection can be quite difficult because the parts of this process are not independent:

1. Statement of the problem. The most important problem-specific data is the client's description of the problem. This initial statement is supplemented by a variety of

questions that give the physician a more complete picture of the patient's problems. This account forms the medical history.

2. Data collection. The medical history is followed by a physical examination and laboratory tests to complete the clinical data base. The results of these three modes of investigation (the medical history, the physical examination, and laboratory testing) then form the patient's medical record. It is estimated that the medical history contributes about 60% of the information necessary to reach a diagnosis, whereas the physical examination and laboratory tests each contribute about 20% [Rosenfeld, 1978].

3. Data analysis (diagnosis). Diagnosis is the assignment of an individual case to a class, excluding other assignments. The traditional practice has been to indicate the primary diagnosis suspected, along with alternative hypotheses listed under 'rule-outs'. Weed [1971] emphasizes a 'bottom up' approach: to simply list the findings available at the highest level of abstraction tenable at that point and not be concerned whether or not this reaches the level of abstraction and certainty implied in the notion of diagnosis. With time and further investigation, convergence on a diagnosis will occur. This 'ladder of abstraction' approach emphasizes that diagnosis is the final abstraction; it not only implies *patterns* of findings, but also *mechanisms* that presumably have predictive power for the future course of the disease and for therapy.

4. Establish a management plan. Select a decision based on the goals of the problem. Management has several aspects:
   - planning for further study;
   - extension or completion of the data base;
   - planning for patient education;
   - planning for general or specific therapy;
   - planning for monitoring the progress.

   The management goal is to optimize the prognosis via the choice between alternative courses of action, or to maximize the certainty of the diagnosis given the data and the possible verifiable conclusions. A critical problem is the question what is optimal, due to value judgments. Choosing the 'best' decision takes into account:
   - the risks of complications of tests and treatments;
   - the dollar cost of tests and treatments;
   - the diagnostic value of tests;
   - patient characteristics such as sex, age, general health;
   - possible spontaneous changes of the patient's state (risk of death);
   - the patient's feelings about the desirabilities of possible outcomes; symptomatic vs. definitive therapy.

5. Monitor the progress of the management through iterative data and/or knowledge acquisition: alter the management plan and/or change the monitoring. Data are needed not only for diagnosis but also for management. Once a diagnosis of appropriate certainty

has been established, data are required for determining the stage of the disease as reflected in the diagnosis, for development of baseline data that will permit later monitoring, and for the monitoring phase that reflects the progress of the disease and its response to treatment.

3.1.1. Diagnostic approaches

Murphy [1976] has characterized the physician's approaches to diagnosis as being of four types:

1. The exhaustive approach (Burke: 'mindless completeness'). This approach is often presented as the ideal to medical students. Sometimes indeed it may be necessary to avoid errors. It is not recommended as standard procedure, because it is too time-consuming and costly.
2. The Gestalt approach, 'the total is more than the sum of its parts'. 'Intuition' and 'hunches' are important, not conscious reasoning. Solutions are reached instantaneously. This approach is related to pattern recognition: humans are superb pattern matchers. But pattern matching is difficult to codify, since it is difficult to identify the importance of pattern features critical for recognition; some elements are difficult to articulate and often elude the non-expert observer.
3. The algorithmic or multiple branching approach. Devise rules as unambiguous guides to decision and action (flow charts). Completeness is difficult except in simple cases.
4. The hypothetico-deductive approach: first form initial hypotheses, then acquire the critical observations to confirm or reject each of the alternative hypotheses. This approach is structured; it pervades the data collection process and is very individual-oriented (both patient and physician). It relies to a significant degree on the physician's understanding of mechanism[1], that is, on his mental model of the pathophysiologic events through which various combinations and patterns of associated findings are produced. According to Murphy, this approach is mostly used in non-routine cases.

The hypothetico-deductive approach seems most suitable for an expert system. The exhaustive approach is unsuited, because in most cases it will be impossible to collect all necessary knowledge. The Gestalt approach is unsuited, because hunches are too vague to describe in an explicit manner. The algorithmic approach is not sufficiently general, because it demands a completely deterministic sequencing through the knowledge. This leaves the hypothetico-deductive approach. For an expert system every case is non-routine, but if the problem domain is sufficiently restricted, it should be possible to enumerate all possible hypotheses.

---

[1] Sometimes such understanding is missing, and associations produced by clinical experience have to be used.

According to Medawar's hypothetico-deductive scheme [Medawar, 1969], diagnosis proceeds in two steps:

1. The hypothesis or idea originates in the 'prepared mind'; this is basically a nonlogical process.
2. The deductive part is usually an experiment to test the hypothesis, a rigorously logical process.

According to Burke [1978], the physician seeks virtual certainty, at least until, on deliberate reflection, he concludes that further pursuit of certainty is unwarranted. Certainty can only be approached, however, in the second step. In the first step, there are two problems. First, it is impossible to know for sure that no hypothesis has been missed. Second, the formulation of a hypothesis presents problems. In SIMPLEXYS, each hypothesis is translated into the pattern 'is X true?' where X is some elementary or higher level concept. All concepts must finally rest on perceived sensor data or *facts*. But how to formulate beliefs (hypotheses) about those facts and how to establish their truth is an unclear, imprecise, intuitive process which calls for much human ingenuity. The high level concept 'the heart rate is stable' is an example. Acquiring the heart rate will usually be no problem; standard monitors can provide the heart rate as a sequence of numbers with sufficient resolution at a sufficiently high rate. Intuitively it is clear what is meant by the concept: the heart rate is regular, its variability is small. But *how* small? Compared to what? The problem is how to *implement* such a concept based on observables only. Many alternatives seem possible. This *synthesis* of new, higher level concepts is an intuitive process.

There is no single 'correct' implementation of a concept. The final test of an implementation is its utility. The chosen implementation of the concept is an *instrument*. If it yields the expected results, it must be considered correct. It then becomes unimportant what the actual implementation of the concept is. As a result, the data underlying the concept have become unimportant; we have moved up in the 'ladder of abstraction' [Weed, 1971] and can reason in terms of the higher level concept only; its *analysis* is fully defined and can therefore proceed automatically.

3.1.2. Protocols

The quality of care becomes an ever increasing concern [Shuman, 1982]. In order to quantify quality, it must be measured. Protocols[1], clinical algorithms[2], are the measurement devices. The yardstick is their specification of what high quality care is. Here, we approach

---

[1] The SIMPLEXYS protocol is a generalization of the protocol described here.
[2] 'Clinical algorithms specify the order of activity, the extent of activity, limits of variations that may occur, acceptable inputs and outputs in any given system. To design an algorithm, one must anticipate all possible conditions and define exactly what will be done in each case' [Skolnick, 1982].

the protocol not so much because we want to *measure* the quality of care, but because a protocol can *specify* what good care is.

According to Donabedian [1978], the first step in exploring quality is the description of diagnostic and therapeutic management as a sequence of activities and strategies. The resulting models, which take the form of algorithms or decision trees, are not only precise and realistic representations of what is considered to be good care but are also a means of testing alternative strategies of care and of confirming or modifying norms. Donabedian cites three reasons for adopting the protocol approach:

1. It provides a basis for formulating criteria for assessing the quality of care.
2. It provides an educational tool for specifying and communicating the operations that constitute clinical judgment.
3. It reveals the deficiencies in current information, identifies those which are most critical, and suggests research needed to obtain the required information.

A protocol begins with a defined medical problem. For that problem, the protocol does six things [Pass et al, 1982]:

1. It delimits the problem by defining which patients with that particular problem can be managed appropriately using the protocol.
2. It indicates the specific data (history questions, elements of the physical examination and laboratory tests) which need to be collected in order to manage the problem.
3. It sequences the acquisition of data by including logic rules that individualize the clinical data according to the particular characteristics of the patient, such as age, sex, medical history, current medications, and characteristics of the patient's complaints.
4. It specifically indicates the clinical findings which are serious enough to require referral to, or consultation with, a specialist.
5. It includes precise rules for arriving at a diagnostic impression and for making management decisions such as prescribing therapy.
6. It can be filed in the record and serve as a progress note.

The protocol logic is often displayed as flow charts, a set of decision rules or a decision table, or in some special format, e.g. combined with a data collection checklist, but increasingly as a set of computer frames, where the computer guides the progress through the protocol (protocols as computerized consultants).

The protocol's *scope* must be appropriate. It can never be complete; it is necessary to define the boundaries where referral becomes necessary. A too narrow scope wastes the user's abilities, but a too wide scope can be dangerous.

In writing a protocol, these steps can be distinguished [Pass et al, 1982]:

- Identify the patients for whom the protocol is intended.
- Identify the cluster of related complaints to be handled by the protocol.
- Identify the diseases or conditions which may cause those complaints.
- Characterize each disease or condition in terms of 1) its likelihood of occurrence, 2) whether it is treatable if found, 3) the seriousness of its consequences, 4) the clinical skills and resources required to identify it.
- Based on the above considerations, decide which of the possible causes of the patient's problem the protocol 1) cannot afford to miss, 2) should refer, 3) can leave to the protocol user, 4) can ignore.
- Identify the signs, symptoms and laboratory tests most likely to be useful in diagnosing the conditions which must be identified if present.
- Write the rules specifying the circumstances under which each of these data is to be collected, each possible diagnosis is to be made, each possible therapy given, and when referrals are to be made.
- Translate these rules into a sequential series of steps to be taken by the user, including all appropriate branching logic.
- Test, correct and validate the protocol through 1) initial peer review, 2) preliminary trials, 3) prospective randomized controlled trials.
- Document the protocol, explaining the medical rationale for each step of the algorithm, and describing the clinical skills required to use it.
- Disseminate the protocol and supporting documentation to the users.

The use of protocols shows striking improvements in the performance of physicians. Even when their use is discontinued, they are better than before, though not as much as when they used the protocol.

3.1.3. Monitoring

'Although the word *monitoring* is often thought to refer to the act of obtaining a measurement, the proper use of the term in anesthesia is to observe *and* control. That is, monitoring is both the obtaining and the use of information' [Ream, 1982].

Monitoring is patient management, with an added critical factor: time. Monitoring is necessary if the patient's physiology is or can become unstable. In intensive care units, the accent is on observation, diagnosis. In the operating room, therapy is the prime concern, surgery, and with it anesthesia. But this therapy is hazardous. Monitoring is the most complex in the operating room during the practice of anesthesia. That is why we focus on it here. Gravenstein [1979]:

> 'Nowhere in medicine - indeed, nowhere at all - is man expected to live through more drastic and dangerous invasion of body and chemistry than in the operating room. Anesthetic drugs not only dissolve in the patient's brain cells to produce amnesia and analgesia [which is intended], but they are also absorbed in the body [which is not

intended] where they affect many functions of the central and autonomic nervous systems, the ganglia, heart, liver, kidneys, muscle and marrow. Literally every organ is touched and often weakened, inhibited, unbalanced and disturbed.
To these anesthetic effects we must add the action of muscle relaxants, the sequelae of unnatural positions, the imposition of artificial ventilation generating unphysiologic intrathoracic pressures and frequently gas tensions in lungs and blood, the surgeon's knife and his assistant's elbow resting heavily on the patient's chest, the loss of blood, the infusion of artificial fluids and old blood with unfriendly preservatives, the imposition of disfavored temperatures and the fasting state. Putting all these together, we gain a faint image of what actually happens, thousands of times daily, in hospitals throughout the world. Most patients survive it all. We are a hardy lot!
Some patients do not survive. Some suffer permanent damage. Some survive, but with impaired brain function. You will claim I exaggerate. Really, only a small percentage die under anesthesia or survive with permanent damage. No one knows the correct figures. Most estimates suggest that less than 0.1 percent come to harm from anesthesia, but most estimates also say that, for the United States alone, the number of anesthetic deaths may be as high as 20,000 a year, maybe more. Even if only a quarter of these were preventable, as many as 5,000 lives could be saved.'

The problems are complex, and time is short. Although one would expect this to complicate patient management, in practice this is not the case. Decisions need to be almost instantaneous. Therefore, choosing a plan for action is left to historical perspective or current consensus. 'The collective choices have been stored in the anesthesiologist's mind-computer during his indoctrination, training and subsequent education' [Bendixen and Duberman, 1982]. Especially in anesthesia, the time available is not sufficient for library reference or rigorously reasoned decisions. Guidelines are developed when there is time available for library work, thinking, and consultation. 'The collective experience and judgment expressed in the consensus seem more valuable for reaching optimal decisions rapidly than the individual experience and judgment of any single anesthesiologist'.

High quality monitoring is impossible without a wide assortment of devices. These must be simple and reliable. They take valuable time to apply and maintain, time taken away from patient care. They are costly, and the information that they provide must be sufficiently useful often enough to justify them. New monitors are usually introduced to solve an immediate clinical problem; even if they are successful, they are usually reluctantly accepted for general use.

More than any other medical specialist, the anesthesiologist follows a protocol-like well-defined procedure [Schneider, 1979]:

'In general, every anesthetist organizes his intraoperative monitoring around a separate graphic chart for each anesthetic he administers. This record has areas in which he enters the patient's name, age, sex, diagnosis and other pertinent preoperative

information; it also has a cross-ruled area for the frequent entry of symbols which represent values for systolic and diastolic blood pressure, respiratory rate and heart rate. A list of the time of administration and the dosages of various drugs is also included. A copy of this record usually finds its way into the anesthetist's files for statistical and billing purposes, while the original becomes the official record of the anesthetic administration and is filed in the patient's hospital record.'

The record guides the acquisition of data, is a receptacle for those data, has a format that eases their interpretation, and serves as documentation of the case.

The life-threatening problems in anesthesia are well known, e.g. Goldstein and Keats [1970], Bendixen and Duberman [1982], Cooper et al [1982], Manteleers [1985], Meijler [1986]. Humans fail far more often than the equipment. Detection of important events by humans is fragmentary at best [Vetter and Julian, 1975], and support in this area is generally considered urgent. But the detection of important events by the current generation of monitoring equipment is poor as well; usually there is a high proportion of false alarms. Therefore such alarms are often turned off, but with serious consequences: 'the ease of disabling ventilator alarms either deliberately or inadvertently is overall the most important contributory factor in injuries' [Cooper and Couvillon, 1983].

In general agreement with Bendixen and Duberman [1982] we conclude:

1. Much of the decision making in anesthesia deals with a relatively small number of regularly recurring events. A methodology for making many of these decisions is clearly discernible and definable, even implementable in a machine.
2. The decision making process consists of:
   - defining the problem; in anesthesia, (parts of) the problem will be very similar in many cases;
   - gathering the information necessary to detect the problem; which information is to be acquired is very much standard in many cases;
   - proposing a choice or course of action to solve the problem; in many cases, the appropriate course of action for a machine would be to issue an alarm, as specific as possible [van der Aa, 1990]; in some cases, the action to be taken will be both unique and mechanizable;
   - checking the proposal to judge its appropriateness; if the machine's proposal proves to be always appropriate, this may lead to further mechanization of part of the therapy (for an example, see chapter 9);
   - effecting an action; some of these actions may be executed by a computer, which can also monitor their progress.
3. Realization of the role of the collective in decision making should lead to greater use of guidelines, policies, and protocols, increasingly in computerized form as 'management assistants'.

4. We can refine our understanding of what the individual anesthesiologist does best and use his decision making skills to the best advantage by incorporating them into the protocol.
5. Bendixen and Duberman [1982]: 'With increasing understanding of the decision-making process, good collective planning, optimum use of technology, and optimum use of human beings, there is little reason why we should not reach or at least come close to our goal of zero anesthetic morbidity and mortality, at least for healthy patients undergoing routine procedures.'

3.2. Machine reasoning

The computer must solve the problems, and it must solve them correctly. Machine logic, however, is one of bits and bytes, of statements or propositions, of some kind of more or less formal mathematical logic. It is not human logic; computers do not *understand* their problems, have no *insight* into their problem domain. Given a problem and a knowledge base, they can only solve the problem if the knowledge to solve it is there and if that knowledge can be accessed. But that knowledge is limited. The computer has an extremely narrow world view, and its evaluation of the situation is in terms of its small world. If the knowledge is not there or not accessible, it may give, by human standards, extremely irrational responses. Incorrect solutions arise when

- the problem is not recognized and therefore either not solved at all or miscategorized into a class for which a solution *is* available;
- decomposition of the problem into the available solvable sub-problem classes is impossible or insufficient; some of the necessary knowledge is missing;
- the inferencing process cannot reach a solution, although the knowledge is there; the proper links to the knowledge are missing;
- no appropriate action can be executed although a solution is reached;
- multiple conclusions and/or actions contradict each other; this may cause chaotic behavior.

These will remain fundamental problems as long as computers cannot be given the same knowledge as an intelligent, experienced, sensitive human. Yet, the computer must not only solve the problems, it must solve them in an expert manner. Whatever the computer's internal problem solving strategy, we want to obtain the solution, and we want it almost instantly. This calls for the computer to have some 'expert' properties:

- It must have the knowledge to solve the problems that must be solved, in particular the appropriate and necessary high level concepts.
- It must know its specialization, and properly recognize problems as either solvable or unsolvable given the existing knowledge. This calls for an exhaustive enumeration of *all* problems that may be encountered. The computer must act like a medical specialist who

solves a problem that *he* cannot solve by referral[1]. In other words, there must be an *explicit class of problems* that need to be referred.
- It must pay attention to the incoming data in two ways. First, it must check whether (some of) the data look suspicious. Second, it must methodically check whether any of the solvable problems occur. Finding suspicious data without an applicable solution may be an indication for a referral.
- It must have the appropriate senses (sensors, input capabilities) to acquire the necessary basic facts that are needed to solve the problems.
- It must recognize the problem's *type* as one of a class that can be solved, and then use the sensor data to solve it. There must be an *internal model* for the problem type. Matching model and data can essentially be done in one of three ways.
  The first method is to pose all possible hypotheses, i.e. try to solve all imaginable problems. This exhaustive approach, even if possible, is extremely wasteful of processing time if many of the hypotheses are irrelevant in the actual problem situation.
  The second method is direct *pattern recognition*: the data form an n-dimensional point within the confines of the stereotype. The data define the type. This 'Gestalt' approach can possibly be applied in simple cases[2].
  The third method is to *be prepared*: expect the problem to be one of a limited *context-dependent* set. Establish the proper *context*, pose a limited number of hypotheses and confirm and/or reject each hypothesis. The context depends on existing or previously acquired knowledge.
- Experts make more inferences in problems they know well. The computer must use the knowledge to generate more knowledge in a 'working forward' strategy. This is akin to 'direct recognition'.
- It must connect appropriate, immediate *actions* (procedural knowledge) to the conclusions. Conclusions are no good if they do not lead to responses. If no appropriate actions are available, no conclusions about them are needed.
- It must provide *consistent* responses. Experts are consistent, and their consistency generates trust.

3.3. The SIMPLEXYS problem solving methodology

In accordance with the above, SIMPLEXYS will have to provide an implementation of a combination of the hypothetico-deductive scheme and the protocol approach, which will allow the knowledge base designer to:

- define a formal description of the possible *problem solving contexts* and the ways in which they are related; this formal description is implemented in STATE rules (see section 4.9), which identify the context, and a *protocol* (see section 5.3.2.4 and chapter 7), which identifies the relations between problem solving contexts;

---

[1] If he does not refer, he does not solve the problem but misses it.
[2] Neural networks take this approach.

- define a formal method to *traverse the protocol* and *establish the prevailing problem solving context*; this method is formally described in ON statements (see section 5.2.4);
- define the possible *hypotheses*; the formal description of a hypothesis is a *rule*;
- define a method to *link hypotheses to a context*; this is achieved by specifying a rule to be a context's *goal* or *goal rule*;
- define a method to *test a hypothesis*, including how to acquire any data which the hypothesis needs and how to present the outcome of the test; in SIMPLEXYS this method is implicit: the *Inference Engine* (see section 5.5) automatically evaluates all goal rules.

To solve a problem, several types of knowledge must be available in the computer. Declarative knowledge consists of elementary facts (primitives) and higher level concepts; in SIMPLEXYS, both are implemented by *rules*[1]. Episodic knowledge concerns the time at which events occurred. It seems natural to link a fact with the time at which it occurred, and this is what SIMPLEXYS does: for each concept it remembers the time when it last occurred, so that questions like 'how long has the heart rate been stable?' can be asked. Procedural knowledge is knowledge about how to *do* something. SIMPLEXYS implements actions as Pascal or C procedures. Scripts [Winston, 1977; Feltovich and Barrows, 1984; Cantor and Kihlstrom, 1985] are the most interesting knowledge structures. Dreyfus [1981]: 'We define a script as a predetermined causal chain of conceptualizations that describe the normal sequence of things in a familiar situation. Thus there is a restaurant script, a classroom script, and so on. Each script has in it a minimum number of players and objects that assume certain roles within the script ... each primitive action given stands for the most important element in a standard set of actions. ... The results of human reasoning are *context dependent*, the structure of memory includes not only the long-term storage organization (what do I know?) but also a current context (what is in focus at the moment?). We believe that this is an important feature of human thought, not an inconvenient limitation.' Just like protocols (section 3.1.2), scripts describe *when* to do *what* and integrate all other knowledge categories. SIMPLEXYS implements the time sequencing of the script as a protocol (chapter 7). Plans are an inherent part of the script. Given a situation, the knowledge is there to describe what must be accomplished (in SIMPLEXYS: the 'goals'), what has happened and what can be expected next.

We now present a complete overview of how these knowledge structures are implemented in SIMPLEXYS, using an application oriented taxonomy. Expert systems frequently make a distinction between 'long term knowledge', which is stored in e.g. rules, and 'short term knowledge', which consists of measurements, results of computations, answers to questions etc., stored in 'working memory'. Due to its real time nature, SIMPLEXYS refines this distinction. Four categories of knowledge can be distinguished. The

---

[1] The reader will have to keep in mind that a 'rule' in SIMPLEXYS is quite different from a rule in a production system.

first category implements fixed knowledge (LTM, long term memory), the other three categories implement more or less volatile knowledge (types of STM, short term memory).

1. *Case-independent fixed knowledge* about the application. This knowledge is implemented as
   a. the *protocol*; this is a description of *when* to use *which* knowledge;
   b. the *rule base*; this is a description of the static semantic and episodic declarative knowledge, and the relations between the individual knowledge chunks, which are called *rules*; among the rules are the *goal rules* (the conclusions to be derived);
   c. a set of *procedures*; procedures implement procedural knowledge, actions to be performed, e.g. to acquire data or to display results; each procedure is attached to a *rule* which denotes the conditions under which the action is to be performed;
   d. a *data memory*; storage locations for data that must be collected, remembered, computed, displayed.

SIMPLEXYS *rules* embody both a *concept* (i.e. an LTM chunk of fixed knowledge), e.g. 'the heart rate is stable', and a (possibly temporary) *assertion* about this concept (an STM structure, e.g. 'it is currently true (or false, or unknown) that the heart rate is stable'. Such a combination is well known from Pascal: a *function* embodies both an *algorithm* (the LTM part) and a *value* (the STM part).

There are six types of SIMPLEXYS rules: fact rules, state rules, memo rules, ask rules, test rules and evaluation rules (see section 5.3.2.3); each type stores a different category of knowledge.

2. *Case-dependent fixed knowledge*, i.e. knowledge about the case which does not change *during* the case. This knowledge is implemented as
   a. *fact rules*, which store fixed facts about the current case, such as the patient's age category.

3. *Medium term knowledge*, which is available once acquired, but may be updated. This knowledge is implemented as
   a. *state rules*, which remember the context;
   b. *memo rules*, which remember earlier results, such as complications that have arisen.

4. *Short term knowledge*, which is to be re-acquired whenever new data are to be analyzed. This knowledge is implemented as
   a. *ask rules*, which store the replies to questions from the user;
   b. *test rules*, which store the results of tests on data;
   c. *evaluation rules*, which store higher level intermediate or final conclusions;
   d. externally supplied *data values*, e.g. patient measurements.

In many small and some larger applications in which SIMPLEXYS was tested, these knowledge representation techniques were demonstrated to be well geared to the type of application envisioned. They were shown to be sufficient and quite versatile.

3.4. Language design

The design of a new programming language can be approached from two directions, a theoretical and a practical. From the theoretical point of view, a language is a tool to build a logical, verifiable 'abstract machine', given some appropriate basic building blocks; this section largely follows Wirth [1976a], the designer of Pascal, who presents some of the requirements for a language and its compiler. Chapter 4 describes the practical approach: explore which basic building blocks are necessary to accomplish tasks in a certain class; the language then seems to design itself.

'The software inflation [the construction of very complex and large programs] has led to a software crisis which has stimulated a search for better methods and tools. This includes the design of adequate system development languages ... The important role of programming languages in the design of large systems is now being recognized. In fact, they are indispensable' [Wirth, 1976a].

The problem is that 'complexity has proven to be a sure winner in attracting customers that are easily impressed by sophisticated gadgets. They haven't sufficiently realized that the additional performance of a complex design is usually much more than offset by its intransparency or even unreliability, difficulty of documentation, likelihood of misapplication, and cost in maintenance. But we shall probably have to wait for a long time, until simplicity will work as a sales argument. To be sure, "simple" must not be equated with "simple-minded" or "unsophisticated", but rather with "systematic" and "uncompromising". A simple design requires much more thought, experience, and sound judgment, the lack of which is so easily disguised in complexity.'

The primary goal of a programming language is to allow the programmer to formulate his thoughts in terms of abstractions suitable to his problem, to build an 'abstract machine'. If the goal of SIMPLEXYS is to encode knowledge, it must have the correct abstractions to do so. The type of problems to solve dictates the choice of the abstractions in SIMPLEXYS (section 3.3).

The second goal is, of course, that the 'abstract machine' must actually run and solve the problems it was designed to solve. The issue is now efficiency, both in terms of code size and speed of execution. But we must avoid forcing the programmer to invent tricks that are based on his knowledge of the way the abstract machine is encoded into the underlying hardware, in order to make his programs have the required efficiency. The abstractions themselves must be efficient.

A third goal is that the 'abstract machine' must be so logically built, that a compiler can 'double-check the legality of the program statements within a well-defined framework of abstraction' [Wirth, 1976a]. In SIMPLEXYS, statement level checking is complete. Not only single program statements must be checked, but preferably also the program as a whole. This is a much more difficult task, because not only syntactic, but semantic knowledge is necessary to accomplish it. In SIMPLEXYS, this latter type of checking is much more thorough than usual, because of the simplicity and formal nature of the constructs.

In particular, interrupts have no place in an abstract machine. Unsophisticated programmers frequently consider interrupts to be 'unexpected events', but such a view is highly inappropriate: no program will ever be able to handle the unexpected. 'An interrupt is a highly machine-oriented concept that allows a single processor to participate in the execution of several concurrent processes. A language should either be devoted to the formulation of strictly sequential algorithms, in which case the interrupt has no place as a concept, or it is designed to express the concurrency of several sequential processes. In this case a suitable form of synchronization operations must be found, but again the interrupt as a concept is inappropriate, because it refers to a processor (machine) instead of a process (conceptual unit of the abstract algorithm)' [Wirth, 1976a]. SIMPLEXYS protocols (chapter 7) express concurrency; interrupts, although not SIMPLEXYS concepts, are easily interfaced with (section 9.3.1).

Wirth [1976a] formulates some criteria for judging a language and its documentation. The first is '*a complete definition without reference to compiler or computer*. Such a definition will inherently be of a rather mathematical nature.'

Yet, it is unavoidable that the language interfaces with the hardware; input/output devices and the filing system must be accessible. According to Wirth, 'language designers are well-advised to provide a facility to delineate modules within which certain device dependent language features are admitted and protected from access from elsewhere in a program'. A Pascal implementation is supported by a small 'run time I/O package'. SIMPLEXYS is similarly supported by an interface to Pascal or C.

'Future languages must provide a *modularization facility which introduces and encapsulates an abstract concept*'. Such a facility 'is instrumental in keeping the size of a language - measured in terms of the number of data types, operators, control structures, etc. - within reasonable bounds'. The SIMPLEXYS rule is such a construct.

Another criterion to judge a language is its *size* in terms of the number of manual pages that is necessary to document it (according to Wirth, a language definition, comprising its syntax specifying the set of well-formed sentences, and its semantics defining the meaning of these sentences, should not extend over 50 pages). 'In programming, we are dealing with complicated issues, and the more complicated an issue, the simpler must be the language to describe it'. Criteria are thus *conciseness and clarity of description*.

Wirth also formulates some requirements for the language's compiler:

- First and foremost, it must be *totally reliable*: it must perform a syntax check against every single rule of the language, translate the program correctly and take care that no incorrect program can crash the compiler. In this sense, the SIMPLEXYS compiler is totally reliable. Semantic checks (e.g. whether the program will come to a halt) are much more difficult. In Pascal, no such checking is done; SIMPLEXYS performs a number of such checks.
- The compiler must also compile at a reasonable speed. This is difficult in SIMPLEXYS, with its extensive overall checking, if the program is large. Therefore the compilation process is split up into several parts. The Rule Compiler performs the compilation and does most syntax checking; it is fast (for an example, see section 9.4.4). Only if the Rule Compiler finds no errors, the Semantic Net Checker starts its checking, which takes a longer time. And only if the Semantics Checker finds no errors, the Petri Net Checker does its checking, which takes a longer time as well.
- The compiler must generate efficient code. This is the most important feature of SIMPLEXYS expert systems: they are extremely efficient compared to other expert systems.
- The execution cost of the code must be reasonably predictable. This is true for SIMPLEXYS expert systems (section 5.7), not for many other real time expert systems (section 2.2.3).
- The compiler should be reasonably compact. The SIMPLEXYS Rule Compiler is small (less than 60 Kbyte), as are its extensions.
- It must provide a simple and effective interface to the environment. The SIMPLEXYS environment is Pascal, and the interface is simple and effective.

Compiler maintenance is very important. For a general purpose programming language, Wirth demands that the compiler be written in its own language. Also, 'the development cost of a compiler should stand in a proper relationship to the advantages gained by the use of the language. *This holds also for individual language features*'. The latter is exactly the reason why SIMPLEXYS is so compact.

In Pascal, language rules that cannot be checked at compile time are verified at run time. In SIMPLEXYS the same is true, but in addition the Rule Compiler generates a number of warnings for questionable or incorrect higher level constructions.

Wirth's overall conclusion is 'that Pascal is a language which already approaches the systems complexity beyond which lies the land of diminishing returns'. The experience with SIMPLEXYS is as yet insufficient for such an evaluation, but preliminary work supports the opinion that the SIMPLEXYS constructs are both suitable and sufficient for the type of problems it means to attack.

4. The origin and evolution of SIMPLEXYS

SIMPLEXYS was not *designed*, it *grew*, out of a need. The need arose when our research was directed to the design of 'intelligent alarms' in anesthesia. Currently, clinical alarm systems are clumsy at best, sometimes even useless. They generate a lot of false alarms, due both to the bad quality of the measured signals in general, and to the fact that those signals are frequently disturbed by willful actions of the medical staff in particular. Thus the need arose to 'filter out' alarms due to artifacts and also to eliminate those alarms that are clinically useless or superfluous.

This proved to require a lot of knowledge about *how* those signals can be disturbed, what clinically useful information they provide and how they are used. This type of knowledge was difficult to incorporate into the data processing algorithms, especially because it was often unclear exactly which knowledge to incorporate: many complex, difficult to program tests were needed. And due to the experimental nature of many of the algorithms, program maintenance became a difficult issue: the representation and implementation of the knowledge into a fixed algorithmic format was often almost impossibly complex, because it was spread out all through the program. Thus we became interested in better techniques to solve these types of problems. Our attention became focussed on expert systems, because they offer a convenient methodology for the representation, implementation and manipulation of knowledge.

In section 1 of this chapter we formulate the requirements that should be met in the applications that we consider. Although the expert systems approach is promising, current products did not meet our demands. In section 2 we investigate how a new product with greater computational efficiency can be attained. Sections 3 through 10 describe the steps that have led to the final product SIMPLEXYS. These sections are not meant to present a design history, but to give the reader some insight into the 'why' of the language features of SIMPLEXYS. Finally, section 11 demonstrates that SIMPLEXYS meets the requirements of what an effective tool should offer.

4.1. Requirements for expert system applications

In a search for an expert system that could be used in the types of application that we had in mind and that could run on a small, inexpensive computer such as a PC, we soon discovered that expert systems research is still a very young science in which most research focusses on theoretical considerations (how to represent knowledge; how to elicit domain knowledge from the experts), and that the products that have reached the market place are mostly immature and reflect the state of the art of several years in the past. Implementations were based mainly on the programming language LISP, a nice and flexible but inefficient

tool, that allows almost anything but needs large computers and even then is slow[1]. Although there are versions of LISP that run on a PC, LISP programs are very slow compared to, say, Pascal programs that perform a similar task; and although there are many versions of LISP that run on workstations, the cost of a workstation is ten times more than that of a PC. We also found that most current expert systems operate interactively, through question and answer sessions. In an interaction, the human who consults the expert system often takes most of the time in considering responses to questions, and thus the speed with which those systems reacts is not too important.

But in real time applications, speed becomes of the utmost importance. Analysis cycle times are short, in the order of 5 seconds. The analysis of the data must be available before those data have become irrelevant, and before the next data arrive: the expert system must be fast enough to keep up with the enormous quantity of rapidly incoming data.

We then formulated these requirements:

1. Our previous applications (e.g. DADS [Meijler, 1986]) were small enough to run on a PC-sized machine; in order to be cost-effective, new expert system based applications must be able to run on a machine as small and inexpensive as a PC, too.
2. Our applications should be fast enough; the expert system should be geared toward real time work.
3. Our applications should have custom-designed user and device interfaces, not something prescribed by the expert system. In particular, a custom interface to peripherals like AD- and DA-converters, RS-232 ports, keyboard and such should be easy. Also, the application, not the expert system tool, should decide what to show on the computer's video display.
4. There will be lots of computations, and these should be as fast as in a standard programming language such as Pascal[2].
5. The expert system should not only be fast *on average*, it should have an easy to estimate worst case response time.
6. The expert system should be easy to program, test, debug and use for persons used to programming in a standard computer language; the people who will be updating and maintaining the expert system are generally not AI researchers.
7. The knowledge base should be easy to read and maintain and serve as a good documentation for the implemented knowledge.

---

[1] Buchanan and Shortliffe [1984] about LISP: 'For a research effort ... we were much more concerned with saving days during programming development than with saving seconds at run time'. In a real time application, however, a one second difference in run time may be very important.

[2] In our opinion, and in that of many educators, Pascal is the best programming language where transfer of ideas is concerned; although a 'C' version of SIMPLEXYS exists as well as a Pascal version, in this dissertation we use Pascal throughout for its better readability.

8. Safety is very important in our type of application. Exhaustive checks of the implemented knowledge, right from the very early design stages, are mandatory; tests should not wait until the final application runs. Safety first, not last!

Our survey of available products indicated that none was suitable. Some appeared able to handle our type of application, but required expensive computers, whereas we wanted our applications to run on something as small as a PC. None seemed to be designed with efficiency in mind. Many had a fixed user interface; we wanted to be able to design our own. And too little information was available to estimate their worst case performance; they were too much a 'black box'. And although many of the commercial packages have, for reasons of efficiency, left LISP behind as their implementation language, their spirit, list searching (or its mathematical counterpart, unification, the basis of Prolog), is still that of LISP.

Safety seemed to be an unimportant issue, too. 'Truth maintenance' is a major theoretical issue, but the *reliability* of expert systems is not. Few products test for semantic errors like conflicts between different chunks of knowledge or infinite loops in the evaluation process. Inconsistencies are hard to avoid in everyday life, yet we want to go as far as possible in the attempt to insure that the end product will be error-free.

4.2. Toward more efficiency

At first, in our search for efficiency, we attempted some approaches different from expert systems. The problem was formulated as follows: given some facts (the input data), design a mechanism to derive all possible conclusions (alarm messages) as fast as possible. We perceived the scheme of figure 4.1.

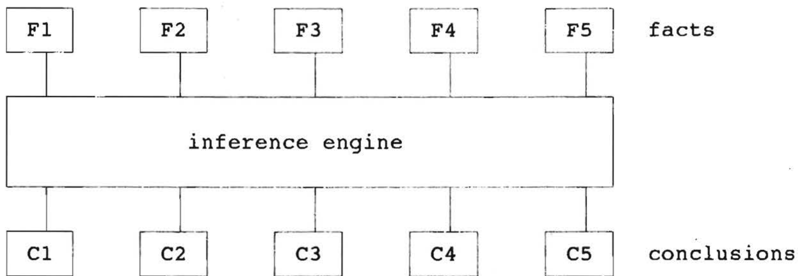


Figure 4.1. An inference engine somehow converts facts to conclusions.

The facts F (i) are given, acquired through e.g. Pascal code, and have 'symbolic' values: either one of *true*, *false* or *unknown*. The conclusions C (i) need to be derived; they, too, are 'symbolic' and can have the values *true*, *false* or *unknown*.

We first investigated the following strategies:

First approach: build a decision table or logical matrix linking the facts with the conclusions. This approach promises maximum speed; in hardware it has taken the shapes of a read-only memory (ROM) and a programmable logic array (PLA). The set of facts forms a pattern or address, that, through a simple lookup, directly generates the output pattern of the conclusions. But this approach proved to be too inflexible; only AND and OR arrays are easy to implement, and we wanted the possibility to realize more complex logical functions.

Second approach: write each conclusion C (i) as a logical function of the facts. Derivation of conclusions is then the evaluation of those functions. Such an approach can be realized in hardware as well, by suitably linking a number of discrete logical building blocks. In software it is just as easy. But we realized that those functions would have many sub-expressions in common, and that repeated evaluations of those sub-expressions should be avoided, at least in a single processor system. So the functions should be 'chunked': common sub-expressions should be discovered, set apart, and evaluated just once. Chunking is essential, too, to retain sufficient insight into the knowledge that is actually implemented. This approach as such is too simplistic; yet, it looked very promising and was always kept in mind. Much of it is, though in a much more sophisticated way, realized in SIMPLEXYS.

We exercized with both approaches, but soon gave up, because the solutions became very convoluted. So we turned to expert systems, our last recourse.

Most expert systems are rule based. Rules, also called production rules, are small 'chunks' of knowledge that are easy to understand and maintain; rule based systems are intuitively appealing. They are also easy to implement. Semantic networks are another choice. They have the advantage of using the relations between concepts in order to optimize searching and to perform some correctness checks. Frames [see e.g. Rich, 1983] are another well known basis, an extension of the semantic network idea. Frames are most useful if the knowledge domain to be implemented has a hierarchical structure. They are more general, but also more difficult to implement than rules. SIMPLEXYS 'rules' borrow ingredients from all these approaches; they look like production rules, they are linked as in a semantic network, and they resemble frames in that they have 'slots' that can specify additional operations and/or actions.

Thus we started what was to be called SIMPLEXYS, not only because we were not satisfied with what was available, but also because we wanted so much insight into the innards of the expert system that we obtained the ability to estimate a worst case performance. Moreover, in a new design we could use every trick we knew to speed up the operation of the resulting expert systems. The enterprise was a challenge for two reasons. First, optimistically, we thought that what we wanted was so simple, that the effort would not be immense. Second, we saw no other alternative.

In a search for efficiency, two roads are open. The first approach promises the most: choose an efficient *algorithm*. A successful choice of algorithm can provide an efficiency

increase of orders of magnitude, e.g. when an algorithm that needs exponential time[1] can be discarded by discovering one that needs linear time[2]. Many of the well known 'basic' algorithms of computer science illuminate this search for ever more efficient algorithms (an example is the large collection of sorting algorithms). The second optimization is an efficient *implementation* of the chosen algorithm. The gains to be earned here are not as large and often require hard labor. Moreover, they frequently require tuning an algorithm to the hardware of the computer it will run on. Up to now, we have expended very little effort in this second type of optimization.

4.3. Start of the Inference Engine

We started with small problems and small systems and tried out and rejected many approaches. The basic idea that over time grew into SIMPLEXYS is due to the following Pascal 'special purpose micro expert system' shown to me by Hajek (Eindhoven University of Technology Computing Center), when he presented his Artificial Intelligence course in 1986. It solved the following problem, adapted from Michie [1980]:

```
Given:   A = unknown    B = true      C = true
         D = true       E = true      F = A and B
         G = C and D    H = E         J = B and G
         K = G and E    X = (F and H) or (J and K)

Wanted:  the plausibility of X
```

Solution (the program, not the plausibility):

```
program infer; {special purpose inference engine}
var A, B, C, D, E: real;
function cand (a, b: real): real; begin cand := a * b end;
function cor  (a, b: real): real; begin cor := a + b - a * b end;
function F: real; begin F := cand (A, B) end;
function G: real; begin G := cand (C, D) end;
function H: real; begin H := E end;
function J: real; begin J := cand (B, G) end;
function K: real; begin K := cand (G, E) end;
function X: real; begin X := cor (cand (F, H), cand (J, K)) end;
begin
  A := 0.5; B := 1.0; C := 1.0; D := 1.0; E := 1.0;
  writeln ('X = ', X)
end.
```

In this little program the 'facts' are implemented as real variables, where 'false' is represented as 0 and 'true' as 1; any value between 0 and 1 represents a plausibility, a fuzzy truth value, a different degree of 'unknown'. The 'rules' of the expert system are

---

[1] An algorithm is exponential time if its run time is an exponential function of the number of elements it has to process.

[2] An algorithm is linear time if its run time is a linear function of the number of elements it has to process.

implemented as real functions that return a 'conclusion', a value between 0 and 1. The operators are real functions as well. The functions 'cand' and 'cor' replace the boolean operators *and* and *or* for this fuzzy logic. A different definition of these functions could be implemented as easily. Evaluation is automatic due to the features of Pascal (recursion). The 'inference engine' is hidden; the inferencing is performed through the standard Pascal evaluation of function X.

This program illustrates many of the ideas behind SIMPLEXYS: the problem (both 'given' and 'wanted') should preferably be as simple as above; the goal or goals ('wanted') must be explicitly stated; the problem must be converted into a compact, efficient 'special purpose inference engine'; rules resemble functions with an LTM part (the function body) and an STM part (the function's result); and inferencing is automatic ('hidden') evaluation of the goal(s).

The main advantages of this program are that it is so simple and compact. There are only two types of rules: one type is implemented as a real variable, the other as a real function that evaluates a logical expression consisting of other rules. And there are only two logical operators, *and* and *or* (due to a conflict with Pascal reserved words the operators cannot have those names).

Another advantage of this type of program is, that the Pascal compiler performs some important correctness checking automatically as long as no functions are declared *forward*, due to the fact that in the program text function B, if it calls function A, must succeed function A. This ensures that no evaluation loops can exist. Logically, such a loop would be a self-referential definition; computationally, recursion would go on indefinitely or until stack overflow.

The main problem with this program is that it solves one problem only. A generalization is easily achieved, however: implement A through E not as real variables, but as functions that allow a user to enter their values, e.g. at a keyboard. Hajek's expert system Quixpert [Hajek, 1988] was set up along these lines. It had two rule types, one type to ask questions, and another type to do an evaluation.

Another problem, not so visible in this short example, is that each question rule asks the same question again each time the answer is needed, and that each evaluation rule does a complete recursion, down to the tips of the problem tree, even if some higher nodes had been evaluated already. Both these problems were solved by storing the rule's conclusion (into an array) and re-using it if it had already been evaluated. Besides convenience (not asking the same thing twice or more), this solution increased the program's efficiency. Now each rule needs to be evaluated once at most: an important step toward a very efficient evaluation procedure. However, since they had become quite complex Pascal functions, Hajek's rules were not very nice looking and understandable anymore, and writing a set of rules was quite an effort. We wanted a well-readable format like:

```
A: 'How plausible is A?'     X: 'X'
ASK                          (F AND H) OR (J AND K)
```

and have some program translate the rules into an efficiently executable format.

ES terminology is not quite standardized. Although in expert systems jargon a *rule* is commonly considered to be a (valueless) 'if-then' *directive*, we call, for want of a better name, and in agreement with Fagan [1980], the 'knowledge chunks' combined in the above lines *rules*, and we refer to the *rules* above as 'rule A' and 'rule X'. In production rule terminology, the *outcome* of the 'evaluation prescription' in the second text line of rule A or X is called the rule's *conclusion* or *consequent*. In frame language, it is called a *variable* or *parameter*, because after evaluation it is assigned a logical *value*.

4.4. Start of the SIMPLEXYS syntax

This approach resulted in an initial syntax and a simple 'Rule Compiler'. In the examples above, rule A was called an ASK rule and rule X an EVAL (evaluation) rule. Compilation of the rules into Pascal functions resulted in very large programs, however, so a more compact representation was devised. The SIMPLEXYS Rule Compiler grew into being, and the compact representation that it delivers is a set of token arrays (see appendix 4). This representation required a 'table walker' program for the interpretation of the tokens and thus the evaluation of the rules; this table walker then became known as the 'Inference Engine' (see section 5.5). The evaluation *method* did not change: a rule was essentially still a function, and it was evaluated by calculating the value of its 'expression' and assigning that value to the rule's conclusion. In expert systems terminology this is called *backward chaining*: evaluate all the necessary sub-rules. We give a very much simplified example. The evaluation rule's expression is:

```
(F AND H) OR (J AND K)
```

The Rule Compiler stores this, in prefix notation, into an array as:

```
OR AND F H AND J K
```

where the operator symbols OR and AND and the rule symbols F, H, J and K now stand for unique *tokens* with a numerical (integer) value. This conversion of an expression into an integer array is called *tokenization* (see appendix 4).

Obtaining the conclusion of a rule is done by a call to function *evalrule*, which evaluates the rule, stores the result for future use and returns the result (the following is a simplification; refer to appendices 4 and 5 for more details):

```
function evalrule (rule);
  var temp;      {local storage in case the call is recursive}
```

```
begin
  value := rule_value [rule];              {recover stored value}
  if value = not_yet_evaluated then         {must evaluate value}
  begin
    case rule_type [rule] of
      ask:  value := ASKval (ruletext [rule]); {ask question}
      eval: begin
              temp := ptr;                 {push current pointer}
              ptr := eval_index [rule];     {expression begin}
              value := evalexp;             {evaluate expression}
              ptr := temp                   {restore old pointer}
            end
    end {case};
    rule_value [rule] := value     {store value for future use}
  end;
  evalrule := value                 {return expression's value}
end;
```

where the following arrays are defined:

```
rule_value [.] stores the rules' conclusions;
rule_type  [.] stores the rules' type;
rule_text  [.] stores the rules' text string;
eval_index [.] stores the rules' index into eval_store;
eval_store [.] stores all tokenized expressions.
```

An evaluation rule's expression is evaluated by function *evalexp*, which interprets the token that *ptr* points at:

```
function evalexp;
begin
  token := eval_store [ptr]; inc (ptr);           {get token}
  case token of
   tNOT: evalexp := apply_not;                    {NOT token}
   tAND: evalexp := apply_and;                    {AND token}
   ...
   1..N: evalexp := evalrule (token);            {rule token}
  end {case}
end;
```

where every operator has a function that applies it, such as

```
function apply_not;
begin
  u := evalexp;
  case u of FA: u := TR; TR: u := FA end;
  apply_not := u
end;

function apply_and;
begin
  u := evalexp; v := evalexp;
  case u of TR: u := v; PO: if v = FA then u := FA end;
  apply_and := u
end;
```

Thus, supported by its Rule Compiler and Inference Engine, a formal SIMPLEXYS syntax developed. An elementary rule consists of two text lines. The first line gives a *symbolic name* to the rule, so that other rules can refer to it[1]. This symbolic name also refers to a variable (rule_value [rule] in the Pascal code above) which will receive the *conclusion* resulting from the rule's application; this conclusion will have a *value* (TR, FA or PO). The first line also gives a *text string*; initially it was the question to be asked, later it became a textual expansion of the symbolic name that could be used both to *compose a question* and to *display the conclusion* resulting from the rule's application. The second line shows how the rule can obtain a conclusion. If it has a keyword like ASK, this line specifies a *method* to acquire a conclusion. If not, it specifies a logical *expression* that must be evaluated, through recursion.

The above discussion explains why SIMPLEXYS rules do not have the familiar production rule if-then format: 'if ... then ...'. Instead, the format is definitional: '... is defined as ...'. A SIMPLEXYS rule thus does not represent an *implication*, but an *assignment*.

For example, the production rule format 'if A and B then C', with logic:

```
   A    B    C
  TR   TR   TR
  TR   FA   **      ** = not defined (C's value is unchanged)
  FA   TR   **
  FA   FA   **
```

is most approximately translated into the SIMPLEXYS format 'C := A and B' with logic:

```
   A    B    C
  TR   TR   TR
  TR   FA   FA
  FA   TR   FA
  FA   FA   FA
```

The translation of the SIMPLEXYS format 'C := A and B' into something like production rule format is: 'if A and B then C else if not (A and B) then not C'. In production rule format, however, a negated conclusion 'not C' does not exist as a production.

The production rule if-then format belongs to two-valued logic, where default values are *false*: C is 'known' to be false unless it can be proved that C is true. The SIMPLEXYS logic is 'richer'; the default value for C is *unknown*, and inferencing tries to establish whether C is true or false; hopefully, C is not both true *and* false: that would be a contradiction. This distinction has been characterized as 'closed world' (in principle everything is known, as in Prolog) versus 'open world' (in principle everything is unknown, as in SIMPLEXYS).

---

[1] The Rule Compiler replaces each symbolic name by an index that is used to access arrays.

4.5. Adding THELSEs

Another Quixpert feature was gladly adopted: multiple conclusions from one rule. Quixpert's philosophy was, due to its nature (the only primitive rule type asks questions), to try to minimize the number of questions asked. Thus, when classifying animals[1] the program reached the conclusion that the animal was a mammal, it could not also be a bird, an insect etc. Thus a rule could look something like:

```
MAMMAL: 'it is a mammal'
... {some evaluation}
IF SO THEN FALSE: BIRD, INSECT
```

The *third* line of this rule mentions a number of rules that get their conclusions essentially free, without computational effort. As soon as MAMMAL becomes true, the conclusion 'false' is assigned to rules BIRD and INSECT. These latter rules need not be evaluated, nor any other rules whose evaluation would be necessary to evaluate them. In correct SIMPLEXYS syntax the third line would read:

```
THEN FA: BIRD, INSECT
```

which would be tokenized as

```
THENFA BIRD THENFA INSECT
```

where the symbol THENFA and the rule symbols BIRD and INSECT again stand for tokens with a numerical value.

The THEN specifies, that the assignment should only take place if the rule evaluates to TR, but other combinations also exist, such as:

```
ELSE PO: assign PO to other rules if the conclusion is FA
IFPO TR: assign TR to other rules if the conclusion is PO
```

These constructs, THENs, ELSEs and IFPOs, collectively called THELSEs, provide another increase in efficiency, because they help us to skip the evaluation of parts of the knowledge base that suddenly turn out to be not relevant or, just the other way round, evidential.

Note, that a THEN TR denotes an *implication*; the THELSE mechanism offers much more than just an implication, however[2].

---

[1] For some reason, possibly because everyone is considered an expert on animals, classifying animals is a popular example in AI texts.
[2] These constructs resemble the inhibitory and excitatory links in what are called neural networks, connectionist models or Boltzmann machines [Ackley et al, 1985].

Definition: A *THELSE* is a construct that specifies that a certain action (e.g. the assignment of a certain conclusion to an other rule) must be performed if the rule that it belongs to obtains a certain matching conclusion; the action is a by-product or 'side effect' of the rule's evaluation.

Definition: A *matching THELSE* is a THEN if the rule's conclusion is TR, an ELSE if the rule's conclusion is FA and an IFPO if the rule's conclusion is PO. THELSEs are executed only if they are matching.

Note that these constructs are *optional*; they are not strictly necessary in a knowledge base. When added, they provide the *cross-links* between chunks of knowledge that make expert reasoning so much faster than novice performance[1].

4.6. Limitation to ternary logic

Another efficiency increase is realized by discarding the fuzzy logic type where all values between 0 (false) and 1 (true) were allowed, and introducing a simpler logic with only the values TR (true), FA (false) and PO (possible; not provably true, not provably false, i.e. unknown)[2]; Quixpert offered a similar option by restricting 'logical values' to only 1, 0 and 0.5. The reasons for this decision mainly depend on:

1. Theoretical arguments: reasoning with Quixpert's logic is semantically not clear and not intuitively obvious. The argument proceeds as follows: Fuzzy algebra demands not only *numbers* for the truth values but also numbers for the *correlations* between all variables. An example will clarify this. Assume an expression like:

   ```
   x := a and b and ... and j, where a = b = ... = j = 0.9.
   ```

   Now probability calculus shows, that the computed probability value of x will depend on the correlation between the variables a, b, ..., j:

   ```
         independence:          x ~ 0.1: almost certainly false;
      full dependence:          x = 0.9: almost certainly true;
   partial dependence:  0.1 < x < 0.9: almost fully unknown.
   ```

   In practice, the correlations are mostly unknown, and even if known, it will usually be too much effort to collect and provide these numbers, which will only be estimates anyway. Thus, computed results would fake an unrealistic measure of precision; in fact, they

---

[1] An abundant number of THELSEs may complicate the maintenance and update of the knowledge base, however.

[2] 'Die Wahrheit der Tautologie ist gewiß, des Satzes möglich, der Kontradiktion unmöglich. Gewiß, möglich, unmöglich: Hier haben wir das Anzeichen jener Gradation, die wir in der Wahrscheinlichkeitslehre brauchen.' Wittgenstein, Tractatus logico-philosophicus 4.464.

would be much fuzzier, much more imprecise than they pretend to be (see also section 2.1.3.1).

2. Computational arguments: reasoning with Quixpert's logic is computationally expensive, especially on a PC (additions and multiplications of reals take longer than a simple lookup, even with a mathematical co-processor), and thus incompatible with the goal of the greatest possible efficiency.

3. Practical arguments: in a real time stand alone system, the final decisions will always be to either perform some action or not. This calls for a yes/no type of logic, not a probabilistic one.

4. User misunderstanding: users had difficulty assigning fuzzy truth values to answers they would really like to give, such as 'usually', 'in most cases', 'almost always'. These can perhaps be handled by more extended types of logic, but we thought that the introduction of certainty *ranges* or other forms of fuzzy logic would confuse our generally formal-logic-naive users even more. Besides, these logics are usually computationally very inefficient. It was therefore decided to choose a simple type of logic, with only the values *true*, *false* and *unknown*, in which uncertainty can then be handled by the rules[1].

Uncertainty is a fact of life. But the kind of uncertainty that SIMPLEXYS deals with is a very special one: it is *uncertainty due to missing information*. It can be quite a predicament if vital information is missing; probability theory, fuzzy logic and such do not come to the rescue if a yes/no type of action is called for. But in many cases the missing information is not vital. Either it can be found in a different way, or it is used in such combinations with known information, that missing it is innocuous because a firm conclusion can be reached anyway. Thus the best interpretation of the truth value PO is: 'I do not have that information', and the best description of the purpose of inferencing is the attempt to reach a firm answer despite missing pieces of information.

4.7. Adding an interface with the outside world

In our type of real time applications, questions are not the only source of information; generally, questions ought to be avoided if at all possible. In addition, there ought to be a rule type to test externally supplied data. In the TEST rule the actual test is done using Pascal code.

The following rule assumes that a device delivers a number HR, which represents a heart rate, and a flag (boolean) VALID, which represents the physiological validity of that number.

---

[1] Experience shows that in SIMPLEXYS knowledge bases even the value *unknown* is seldomly used.

```
HR_NORMAL: 'the heart rate is in the normal range'
TEST
  if not VALID then
    TEST := PO
  else if (HR > lowerbound) and (HR < upperbound) then
    TEST := TR
  else
    TEST := FA
ENDTEST
```

The keyword TEST in the rule's second line specifies a rule of type TEST, a 'hook' into Pascal. The text between the words TEST and ENDTEST is Pascal code that should somehow assign a value of type bool (TR, FA or PO) to Pascal variable TEST; if no such assignment is done, FA will be returned as the default conclusion[1]. Some extra code will subsequently take care of the assignment of the value of TEST to the rule. This takes care of links with the outside world, since the Pascal code can do that. These links are an essential ingredient to test data, to provide an *input* from the outside world.

A simplification of the TEST rule is the BTEST (for Boolean TEST) rule, which can only return TR or FA. This 'syntactic sugar' makes many rules much easier to read. If no validity check is necessary, a simpler alternative syntax for the above rule is:

```
HR_NORMAL: 'the heart rate is in the normal range'
BTEST (HR > lowerbound) and (HR < upperbound)
```

where the text following BTEST contains a Pascal *boolean expression*.

An *output* link was provided by the so-called DO codes. DO's can execute any Pascal code, e.g. to provide for actions such as displaying a result or storing or updating of variables. The previous example expanded:

```
HR_NORMAL: 'the heart rate is in the normal range'
BTEST (HR > lowerbound) and (HR < upperbound)
ELSE DO writeln ('the heart rate is abnormal')
```

The ELSE specifies, as usual, that the code should be executed only when the rule evaluates to FA.

4.8. Adding forward chaining

Thus far, a conclusion that must be evaluated (which we started to call a *goal rule*) can only be derived by recursively evaluating its constituent rules, if any, until the recursion ends in the primitive rules. This type of evaluation is called backward chaining. Forward chaining works the other way around: whenever a rule is evaluated, be it primitive or not, it may

[1] Therefore the 'else TEST := FA' part in the rule above is superfluous.

become necessary, depending on the outcome, to evaluate other rules as well. Thus other rules can be given conclusions (likewise recursively) just as if one evaluation had more than one conclusion. The THELSEs can be considered a form of forward chaining: when a rule conclusion becomes known, other rule conclusions may become known too. True forward chaining is different, though: when a rule conclusion becomes known, other rule conclusions may need to be *evaluated*, not *assigned* a fixed conclusion. Forward and backward chaining complement each other. Sometimes, the one is more suitable, sometimes the other. We felt that SIMPLEXYS should allow both.

The basic mechanism for forward chaining was there already, provided by the THELSEs. We just needed something like the last line of the following rule:

```
HR_ABNORMAL: 'the heart rate is not in the normal range'
BTEST (HR < lowerbound) or (HR > upperbound)
THEN GOAL: HR_ALARM
```

The THEN has the usual meaning. The combination THEN GOAL states, that if HR_ABNORMAL becomes true, rule HR_ALARM should be *evaluated*. It is possible, but not certain, that an alarm should be given; that depends on other conditions. Rule HR_ALARM can then check those conditions and, depending on the outcome, either issue the alarm or not by means of a DO. THELSE GOALs implement *forward chaining*.

The following example solves

```
X = (F AND H) OR (J AND K)
```

by forward chaining. Forward chaining needs a start rule; in this example the start rule is called S, and its conclusion is TR. Since in this example rule types are unimportant, we use a simplified notation.

```
S THEN GOAL: F
F THEN GOAL: H
H THEN TR:   X, ELSE GOAL: J
J THEN GOAL: K
K THEN TR:   X
```

In SIMPLEXYS, rules that start a forward chain are most often STATE rules (see next section).

4.9. MEMO rules, STATE rules and rule histories

For some time SIMPLEXYS had just been a toy and an exercise in learning about expert systems. A lot of small example programs had been designed and successfully run, but some questions remained. The most important one was how to sequence a series of analyses in

applications like patient monitoring, where new data arrive every few seconds (see figure 4.2).

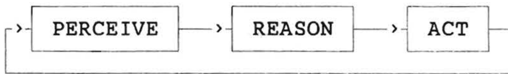


Figure 4.2. Sequencing analyses (from Woods [1986]).

In order to decide whether SIMPLEXYS was a suitable tool for the design of this type of expert system, we looked around for an existing expert system that could be emulated in and translated to SIMPLEXYS. For several reasons, we chose Fagan's Ventilator Monitor [Fagan, 1980; see also section 2.3]. First of all, it represented an example of the type of application we had in mind: monitoring artificially ventilated intensive care patients for respiratory problems. Second, because the problem was excellently described: Fagan's dissertation gave a complete and easy to understand listing of all his rules; we did not have to collect the knowledge and could focus on implementation issues. Third, because VM was written in LISP, so we could try out whether SIMPLEXYS was really much faster[1]. Most of Fagan's rules[2] were easy to translate into SIMPLEXYS. Yet, SIMPLEXYS needed a few more features. The VM experiment had concentrated on how to handle context (is the patient being monitored or not? is the patient on artificial respiration or not?) and on time (how long has the patient's ventilation been assisted?). In a real time process, the situation (the context) changes continuously and the context determines how to interpret the observations (figure 4.3), i.e. which rules must be evaluated. In SIMPLEXYS, there was no way yet to store context information or to remember earlier results. Thus the MEMO and STATE rules and the 'rule histories' were added.

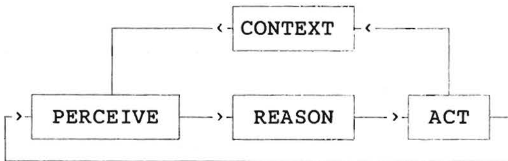


Figure 4.3. Sequencing expectancy dependent analyses.

The task of a MEMO rule is to provide a memory function from one run to the next.

---

[1] Since we were unable to obtain Fagan's data, this comparison was never effected.
[2] These rules are implemented in LISP but documented in pseudo-English. In some instances the use of this less exact pseudo-English and the absence of LISP code presented us with the problem of what exactly was meant. In those cases we used our own expertise in an attempt to recreate the exact interpretation.

Definition: A *run* is the process that derives all necessary conclusions given a certain *fixed* set of inputs (i.e. these inputs belong together and do not change during the run).

The assignment to a MEMO conclusion is by an INITIALLY[1] or by a THELSE, and 'evaluation' of a MEMO rule is just recovering its stored conclusion. An example of a MEMO rule:

```
NMN: 'the patient is not being monitored'
MEMO
INITIALLY TR
```

and an example of a rule that updates the MEMO rule's conclusion is:

```
MTR: 'the monitoring devices are operational'
TEST ...
THEN FA: NMN {if the TEST succeeds, NMN is to become FA}
THEN TR: MON {if the TEST succeeds, MON is to become TR}
```

The assignment of a new conclusion to the MEMO rule cannot take place *during* a run; other rules may already have used the value of the conclusion, and, in order to guarantee consistency, this value must remain constant during a run. The new conclusion is therefore remembered, and takes effect in the next run (see figure 4.4).

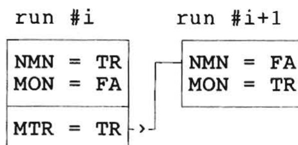


Figure 4.4. Conclusions of MEMO rules do not change during a run. The new conclusion becomes effective in the next run.

The third line of rule NMN shows another addition to the SIMPLEXYS syntax. It specifies, that rule NMN has the conclusion TR when the expert system starts up, i.e. that initially the patient is known not to be monitored.

MEMO rules proved to be a welcome addition, but using MEMO rules to store context information led to rule sets that became difficult to comprehend. Some MEMO rules just stored a temporary result (e.g. 'T-piece occluded'), while others were part of the protocol's time sequencing. To separate these two important functions, STATE rules were added to explicitly denote and remember contexts[2], and ON statements to implement context

---

[1] Assignment of an initial value when the expert system starts up.

[2] Allen [1983]: ... the representation should facilitate plausible inferences of the form 'if fact P is true now, it will remain true until noticed otherwise'.

switches. This separation of functions is mainly 'syntactic sugar', but it proved important: it greatly improves the understandability of the knowledge base, the analysis of problems and the checks that can be performed by the Rule Compiler. So the above rule became:

```
NMN: 'the patient is not being monitored'
STATE
INITIALLY TR    {NMN is the initial context}
THEN GOAL: ... {conclusion(s) to derive}
```

STATE rules embody the concept 'as long as'; here: *as long as* the patient is not being monitored, the goal is to derive a certain set of conclusions.

A context switch was denoted as:

```
ON NMN_to_VOL FROM NMN TO VOL
```

where *trigger rule* NMN_TO_VOL tests whether a context switch from NMN to VOL must be done (see figure 2.1). Trigger rules (or simply triggers) embody the concept 'as soon as'; *as soon as* the conditions embodied in the trigger rule NMN_to_VOL are satisfied, NMN becomes false and VOL becomes true. Code was added to the Inference Engine to actually actuate the context switches: if NMN was true, then if NMN_TO_VOL also evaluated to TR, then NMN was to be set to FA and VOL to TR. In chapter 7 we will show a *Petri net representation* of contexts and context switches.

Definition: A *context* is, at a certain moment of time, that set of STATE rules that have conclusion TR.

Definition: A *context switch* is the process, that, through an evaluation of trigger rules, decides whether some STATE rules should become FA and others TR.

Definition: A *trigger rule* is a rule (the one following the ON in an ON statement), that specifies that, if a certain set of STATE rules (the FROM-set) all have conclusion TR *and* if the trigger rule evaluates to TR, then all those STATE rules obtain conclusion FA, while all rules in another set of STATE rules (the TO-set) obtain conclusion TR.

Definition: A *protocol* is the set of all possible context switches, i.e. the combination of all ON statements.

Although no formal comparison study has been conducted yet, we currently believe that most, if not all, questions that can be answered by a non-probabilistic temporal logic [Allen, 1983; Tsang, 1987] can also be answered by a SIMPLEXYS protocol.

Goals or goal rules are special only in that those rules (usually of type EVAL) are explicitly defined as goals. The THELSE GOAL construct is a convenient method to define the goals and to start the inferencing process.

Most often, *primary goals* are connected to a STATE rule by a THEN GOAL; in that case, they are evaluated only when that STATE rule is TR (in other words: is active, belongs to the active context). However, primary goals can also be connected to a STATE rule by an ELSE GOAL; in that case they are evaluated whenever that STATE rule has the conclusion FA.

Why this is so needs some explanation. Initially, the idea was that the *active context* should dictate all that should be done, in other words that STATE rules only ought to have THENs, not ELSEs (STATE rules are forbidden to have IFPOs). This would make the meaning of the 'protocol' more perspicacious: do such and such only under such and such conditions. Yet we have removed the restriction. First, because it comes in handy once in a while to have an ELSE GOAL from a STATE rule; and second, because the restriction can be circumvented by clever programming anyway.

Secondary goals are goal rules connected to other rules in a THELSE GOAL part. They are evaluated as a consequence of the evaluation of the rule that they are connected with; forward chaining is realized this way[1].

Definition: A *goal* is a rule which is the argument of a THELSE GOAL of a rule.

Definition: A *primary goal* is a rule which is an argument of a THELSE GOAL of a STATE rule.

Definition: A *secondary goal* is a rule which is the argument of a THELSE GOAL of a rule which obtains a conclusion during the evaluation of a primary goal.

VM also had rules that checked whether a certain condition had existed for a certain time. SIMPLEXYS needed a similar construct. Thus the concept of rule histories was added, as well as appropriate additions to syntax, Rule Compiler and Inference Engine. An example: the term

```
NMN > (120)
```

in an evaluation expression has the semantics 'rule NMN is now true *and* it has been true for at least 120 seconds', with outcome one of TR, FA or PO, as usual. The term

```
HRSTABLE > (30 * minutes)
```

---

[1] Because a THELSE GOAL asks for the *evaluation* of goal rules, FACT, MEMO and STATE rules cannot logically be goals.

has the meaning 'the heart rate has been stable for longer than 30 minutes'. Here, 'minutes' is a Pascal constant with value 60. In fact, any valid Pascal numerical expression can be inserted between the '(' and ')', including expressions containing variables[1].

With these additions to SIMPLEXYS, which were easy to implement because the main structures were there already, a paper re-implementation of VM was easily accomplished.

4.10. FACT rules

While monitoring a patient, there are things that never change, such as the patient's age category or whether the patient has a known history of a certain disease. Such data are normally available *before* the monitoring process starts. FACT rules were added to store this type of invariant information.

The idea is as follows. The knowledge in a knowledge base will often contain information about a large class of patients, and thus a lot of rules that depend on and refer to this type of 'factual' information. But the knowledge will be applied to the monitoring of *one* individual patient, not to the whole class. If we insert that 'individual' information into the knowledge base, we obtain an 'individual' knowledge base that may be a lot simpler and more compact than the original one. And thus a lot faster to evaluate. This could yield another major efficiency increase.

Such methods do exist (see section 6.5.2.3.4); they depend on specialization or optimization. A process of semi-symbolic evaluation of EVAL rule expressions can eliminate all references to FACT rules once their conclusions are known, thus simplifying the expression. An example: if A is known to be always true, rule X having the expression A AND B AND C simplifies to B AND C. Another example: if A is known to be always true, rule Y having the expression A AND B simplifies to B; rule Y is superfluous, can be removed, and all references to Y can be replaced by references to B.

The simplification process may in turn lead to a simplified rule with a fixed conclusion. An example: if A is known to be always true, rule Z having the expression A OR B simplifies to true; now rule Z and all references to it can be eliminated as well. Many rules might eventually be completely eliminated[2]. Problems such as Michie's (section 4.3) would totally disappear by 'implosion'.

---

[1] These variables can be manipulated by the THELSE DOs of other rules.

[2] Explanations of the steps taken in 'reasoning' would become very difficult after compaction, however, since many of those steps would have disappeared; it would seem like the expert system 'jumps to conclusions' (but quite correctly, like experts do). But since the compacted net is logically fully equivalent with the original one, explanations, if necessary, could repeat the evaluation using the original net.

4.11. The relation between SIMPLEXYS and production systems

Many expert systems are production systems (see section 2.1.3). Since they are based on a collection of rules of the form

```
if P then X
```

production systems can be said to be *rule oriented.* Frequently more than one rule can lead to the same conclusion. For example, assume that a rule base contains (only) three rules that can yield a conclusion X:

```
if P then X
if Q then X
if R then X
```

SIMPLEXYS can be said to be *conclusion oriented.* It is a characteristic of SIMPLEXYS that every conclusion is connected to one and only one rule. This was achieved by introducing the operator *or*, which allows the above three rules to be combined into one,

```
if P or Q or R then X
```

in which the *or* operator is evaluated conditionally; if P is *true*, neither Q or R needs to be evaluated, just like in the original combination of three rules. Since no other rule can lead to conclusion X, and since the default conclusion for X, if neither P nor Q nor R is true, is false in two-valued logic, the above rule has the meaning:

```
if P or Q or R then X := true else X := false
```

which is equivalent to the SIMPLEXYS-like notation:

```
X : P or Q or R
```

which resembles a definition rather than an implication.

Connecting each possible conclusion to a single rule has a number of advantages, the most important of which is that it becomes much easier to link all rules together in a net (see chapter 6). This linking can *eliminate all searching* for rules which are applicable in a certain situation; it also allows a number of checks on the correctness of the interactions between rules.

Production systems are built on implications and thus are conceptually forward oriented, while SIMPLEXYS is backward or goal oriented. Other features of SIMPLEXYS describe which goals are to be derived under which conditions. The STATE rules provide a mechanism to specify the context, which defines which goals are to be evaluated. And the

ON statements provide a mechanism to specify how the context changes, leading to the evaluation of a different set of goals.

4.12. A first evaluation

The re-implementation of VM had been easy, and we expected the same with the implementation of other systems. Our requirements (section 4.1) were fully met. To reiterate:

- SIMPLEXYS expert systems are easy to program and use; that was demonstrated by several students who built smaller or larger test applications, including the VM re-implementation, which took less than 2 person-weeks for a student who initially knew only the basics of SIMPLEXYS.
- SIMPLEXYS applications are small enough to run on a PC; the storage of the knowledge is very compact. The VM re-implementation (118 rules, 12 ON statements) resulted in an expert system code size (.exe file) of only 58 Kbytes, including the (simple) user interface.
- SIMPLEXYS applications can run on a PC and yet be fast enough; the inferencing method is ideally suited to real time work and the Inference Engine has little overhead. On average, a run of the VM re-implementation was estimated to take approximately 0.5 seconds on an IBM PC-XT (8088 processor) system with a 10 MHz clock.
- The inferencing method (each rule is evaluated at most once) allows a good estimate of the worst case performance (see section 5.7). No run of the VM re-implementation took more than about 0.8 seconds on an IBM PC-XT (8088 processor) system with a 10 MHz clock. The worst case to average run time ratio is small, certifying good processor utilization.
- Computations are efficiently done in Pascal.
- Our applications can have custom-designed user and device interfaces; the easy interface with Pascal allows anything. The VM re-implementation could ask for values or obtain them from a file, and could display numerical, textual and symbolic results.
- The knowledge structures that are available in SIMPLEXYS are well geared to the type of applications. Due to the SIMPLEXYS syntax, the knowledge base is easy to read and maintain and serves as a good documentation for the implemented knowledge.
- The implemented knowledge can be checked thoroughly, starting in the very early design stages. Safety was a major design issue. Chapters 6 and 7 on semantic and protocol checks will amplify this.

5. SIMPLEXYS: a real time expert systems toolbox

This chapter gives a detailed description of SIMPLEXYS. Section 1 gives and discusses an example of a SIMPLEXYS program or 'knowledge base'. Section 2 explains the semantics or 'meaning' of the different syntax elements and how they are used in 'knowledge programs', section 3 shows how SIMPLEXYS programs are structured, and section 4 describes how to program in the SIMPLEXYS language. Section 5 describes the inferencing process. Section 6 gives an overview of the SIMPLEXYS 'tools' that are necessary to build, test and debug expert systems.

The intentions in the development of SIMPLEXYS were efficiency and safety. Some details on the efficiency of execution times of a knowledge base are treated in section 7, while a review of the major efficiency issues is given in section 8. Safety aspects are treated in chapters 6 and 7.

5.1. An example of a SIMPLEXYS program

```
RULES                                                   {RULES section}

RUNNING: 'computing'
STATE                                           {the only context rule}
INITIALLY TR              {TR, thus (part of) the initial context}
THEN GOAL: X              {work to do in this context: evaluate X}

X: 'rule X'
(F and H) or (J and K)        {evaluation rule X's expression}

F: 'rule F'
ASK                       {obtain F's conclusion through asking}

H: 'rule H'
BTEST 4 > 3      {obtain H's conclusion through testing data}

J: 'rule J'
MEMO                          {recover J's previous conclusion}
INITIALLY TR

K: 'rule K'
FACT                            {recover K's (fixed) conclusion}
INITIALLY TR

READY: 'ready'
F and H and J and K     {evaluation rule READY's expression}

PROCESS                                              {PROCESS section}

ON READY FROM RUNNING TO *   {finish as soon as READY is TR}
```

Figure 5.1. A small but complete and correct SIMPLEXYS program.

Chapter 4 has given an informal description of all the SIMPLEXYS syntax elements. Appendix 1 gives a formal definition of SIMPLEXYS as a programming language. An example of a SIMPLEXYS program or 'knowledge base' is given in figure 5.1.

In contrast with most other expert systems, each SIMPLEXYS rule is associated with a unique conclusion (consequent, result)[1]. This allows us to speak of *a rule's conclusion* and to equate *application of rule R* to *evaluation of rule R's conclusion*; we will also use the equivalent shorter terminology *evaluation of rule R* to signify both aspects: the application of the rule *and* the evaluation of its conclusion. This evaluation can refer to the computation of the value of a logical expression (rules X and READY), to the acquisition of the answer to a question (rule F), to the acquisition of the result of a test expressed in Pascal code (rule H), or to the recovery of an earlier stored value (initially, the value *true* has been stored into the conclusions of rules RUNNING, J and K).

The RULES section contains all the rules. STATE rules specify the goals. STATE rule RUNNING, whose initial conclusion was stipulated to be TR, specifies, through its THEN GOAL, that X is the *goal rule*, i.e. the goal is to derive or evaluate conclusion X. Derivation of conclusion X is performed by evaluating rule X's 'evaluation expression'. In this derivation, (at least some of) the conclusions of rules F, H, J and K need to be acquired; these conclusions can be acquired by asking the user, performing a test on data, or by recovering a previously stored conclusion. The goals remain unchanged as long as the conclusion of STATE rule RUNNING has value TR. Thus conclusion X will be derived repeatedly; each derivation of conclusion X constitutes a *run*. The PROCESS section states that rule READY is the *trigger rule*, which must be evaluated as long as RUNNING evaluates to TR. As long as READY evaluates to FA or PO (because at least one of F, H, J and K does not yield conclusion TR), conclusion RUNNING is unchanged and a new run is started. As soon as READY evaluates to TR (because F, H, J and K all yield conclusion TR), the context switch makes STATE rule RUNNING's conclusion FA, and the program comes to an end; the end of a program is reached as soon as none of its STATE rules conclude to TR anymore. This sequencing is shown in figure 5.2.

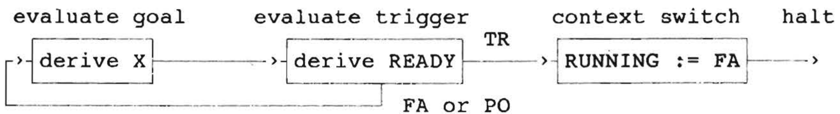


Figure 5.2. Conclusion X, the goal, will be derived repeatedly until conclusion READY, the trigger, evaluates to TR. Then conclusion RUNNING is set to FA and the inferencing halts.

[1] THELSEs provide an additional mechanism. A THELSE lets one rule's conclusion stipulate conclusions associated with other rules without the necessity of actually evaluating those other rules.

The SIMPLEXYS philosophy hinges on the idea that in each run every rule needs to be evaluated just once; a second 'evaluation' is just a look-up of the conclusion of the first. The underlying idea is that the data to be analyzed come from a 'snapshot' of the world; these data have no time skew, 'belong together', and deriving the same conclusion about the same data twice or more times is superfluous and a waste of time.

Instead of coming from a photographic snapshot, we can also view the data as coming from a filled-out form. In contrast with a photographic snapshot, a form's data may be mutually inconsistent. If possible, inconsistencies between the items should of course be discovered; the problem is, however, that this discovery requires deep knowledge and is even then not always possible. The same is true if the data originate from a collection of monitoring devices; a device can be disturbed and generate erroneous results. Yet, here too, deriving the same conclusion about the same data twice or more is superfluous.

The snapshot notion does create problems, however, if a true snapshot (i.e. collecting all the data at the same time instant) is impossible. If a problem must be solved by asking questions, all questions cannot be posed at the same time; moreover, we want to pose only the relevant questions, so whether a question is asked may depend on the answers to other questions. Likewise, a filled-out form need not be complete; in some cases, some entries may be irrelevant. We must now assume that the data supplier keeps in mind that all questions 'belong together' and are about one specific problem. But again, it is important to try to discover inconsistencies in the data supplied.

5.2. Elements of the SIMPLEXYS syntax

The SIMPLEXYS programming language is based on a few important concepts that will be described before entering into details.

5.2.1. Rule's conclusions

SIMPLEXYS is based on a three-valued logic [see e.g. Mamdani and Efstathiou, 1988]. Conclusions have values of Pascal type *bool* (pseudo-boolean). Operations on conclusions expect and return variables of type bool:

type bool = (TR, FA, PO) {true, false, unknown}

TR: the variable's value is true;
FA: the variable's value is false;
PO: the variable's value is unknown ('possible'); it is impossible to show that the variable's value is either true or false; it could be either false or true; *there is currently no information that allows us to decide whether the value is true or false.*

In some expert systems other types often exist, e.g. enumerated types; variable AGE might have a value of either INFANT, BABY, CHILD, ADULT or AGED[1]. Variables of this type do not exist in SIMPLEXYS but they can be translated into a number of variables of type bool, e.g. AGE_is_INFANT, AGE_is_ADULT etc. (which in this case are mutually exclusive, too). Type bool is *sufficient*: it can implement any other finite type.

Rules obtain their conclusion in one of two different ways: by *evaluating* the rule or by *assigning* a conclusion to the rule. A rule is evaluated whenever its conclusion is needed but not yet available. A rule can be assigned a conclusion as a specified side effect of the evaluation of another rule.

A rule also has a *history counter*. At all times, this history counter contains the period, in seconds, during which the rule's conclusion has uninterruptedly been the same. So a rule actually has *two* values, a *bool* value (TR, FA or PO), and a *history* value, an integer number in the range 0 to 2^31 (approx. 2000000000, equivalent to more than 60 years). Thus we will not only be able to answer questions like 'is X true?' but also 'how long has X been true?'.

5.2.2. Rule types

There are 5 primitive rule types:

1. FACT rules denote constants; the conclusion of a FACT rule never changes. FACT rules obtain their conclusion (TR, FA, PO) from known facts, acquired before the inferencing starts, e.g. from a database or from questions to the user. Thereafter these conclusions cannot change again in subsequent runs[2].

2. ASK rules ask questions from the user, who can answer either 'y' for 'yes' (giving a conclusion TR), 'n' for 'no' (giving a conclusion FA) or '?' for 'I do not know' (giving a conclusion PO). ASK rules return a conclusion as soon as the answer is entered on the keyboard. In a real time expert system, therefore, ASK rules should be avoided, but during the design phase they are indispensable[3]. The conclusion is valid during the current run only.

3. TEST rules test externally supplied data through an interface to Pascal. The result of the test is either TR, FA or PO. The conclusion is valid during the current run only.

---

[1] Such variables are quite convenient in an interactive expert system, because they lead to less questions.
[2] FACT rules can be 'optimized away', so that the inference engine never need encounter one; see sections 4.10 and 6.5.2.3.4.
[3] Asking questions without having the system wait for the answers is treated in section 8.1.3.

4. MEMO rules remember results. MEMO rules obtain their conclusion exclusively through other rules' THELSEs. Unlike ASK and TEST rules, MEMO rules keep their conclusions across runs unless a new conclusion is assigned. For an example, see section 4.9.

5. STATE rules denote the current context. Their conclusions are assigned either as an initial conclusion for the first run or via the 'protocol' (a set of ON statements that describe context switches).

   There is only one rule type that combines results of other rules:

6. EVAL (evaluation) rules calculate higher level conclusions, given an expression that combines other rules (either primitive or evaluation rules) using a number of monadic and dyadic operators. The conclusion is valid during the current run only. An example of an evaluation rule:

   TIGER: 'it is a tiger'
   MAMMAL and CARNIVORE and TAWNY and BLACKSTRIPED

   This rule can be read in two ways. First, syntactically, as a sequence of test procedures: 'if the object passes the test for MAMMAL, then the test for CARNIVORE, next the test for TAWNY, and finally the test for BLACKSTRIPED, it will be considered a TIGER'. Second, semantically, as a definition: 'a tiger is a blackstriped, tawny, carnivorous mammal'. If there is a discrepancy between these two readings, either the definition is incorrect, the definition is implemented incorrectly, or both.

5.2.3. The logic of SIMPLEXYS

To evaluate expressions, SIMPLEXYS implements a simple propositional type of logic, comparable to boolean logic. SIMPLEXYS expressions consist of two types of entities, *propositions* and *operators*. Propositions are indicated by 'names', such as p, q, r, animal, normal, etc. Operators are indicated by the 'special symbols' *not*, *and*, *or*, etc. Operators are *monadic* (monadic operators have one argument, e.g. *not* in: not p) or *dyadic* (dyadic operators have two arguments, e.g. *and* in: p and q). Parentheses can be used to form subexpressions. Examples of correct expressions are:

```
p
not poss p
p and (q > (30))
q or s or not t and u
not p or ((q and not poss r) and s)
```

Expression evaluation normally proceeds from left to right, but parentheses can force a different order. History operators have the highest priority. Next in priority are monadic

operators, which all have the same priority; and dyadic operators, which all have the same priority, too, have lowest priority. Thus

```
not A and B or C > (s) and D                                   (A)
```

is logically identical with and evaluated as

```
((((not (A)) and B) or (C > (s))) and D)                       (B)
```

In most logics, *and* is considered to have a higher priority than *or*. In SIMPLEXYS, priority can be forced with parentheses:

```
(not A and B) or (C > (s) and D).                              (C)
```

Thus, in SIMPLEXYS, expression (C) is *not* logically equivalent with (A).

5.2.3.1. Two-valued logic

Many formal logics are two-valued. In a two-valued logic, a logical *value* can be either TR (true) or FA (false). Monadic operators (i.e. operators with one argument) transform a logical value into a new logical value. This is written as:

```
q := op1 (p)          or          q := op1 p
```

where p and q are logical values and op1 is the operator. In two-valued logic, the only meaningful monadic operator is *not*.

Dyadic operators (i.e. operators with two arguments) combine two logical values into a new one. This is written as:

```
r := op2 (p, q)       or          r := p op2 q
```

where p, q and r are logical values and op2 is the operator. Some meaningful dyadic operators are *and*, *or*, and *implies*.

The *names* of the operators currently used in formal logic have a long history; they are identical with and originate from natural language words (mathematical logic can be thought of as a formalization of the grammar of natural language words like 'not', 'and', 'or', 'if ... then ...'). Other choices of operator would also be possible. In fact, only *one* well chosen dyadic operator would be sufficient to implement all others (e.g. *nand* = *not and*, *nor* = *not or* = neither, also called Sheffer's stroke).

5.2.3.2. Three-valued logic

In the type of three-valued logic, that is implemented in SIMPLEXYS, a value can be either TR (true), FA (false) or PO (unknown). Three-valued logic is a better approximation of human reasoning, because in many practical situations it will neither be possible to decide that a proposition is true nor that it is false. If p is any proposition, we can distinguish these three truth values:

p = TR: p has been proved to be true;
p = FA: p has been proved to be false;
p = PO: p has neither been proved to be true nor proved to be false; *at this moment* we do not have the knowledge to decide either that p is true or that p is false.

The logical values TR, FA and PO are not equally satisfying; 'reasoning' will usually attempt to establish a definite answer (TR or FA), even if one or more values in a logical expression are unknown (PO).

In the internals of SIMPLEXYS, invisible to the user, there is a marker or tag UD (for 'undefined'), which often plays a role analogous to a truth value:

p = UD: no attempt has yet been made to establish p's truth value; rule p has not been evaluated yet.

Whenever the Inference Engine encounters p = UD, this encounter implies that p's truth value is needed and will immediately and automatically lead to an attempt to show that either p is true or that p is false, possibly resulting in the conclusion that the knowledge to prove either is missing. Thus UD is not a truth value proper, and the value UD does not occur in proofs. When tracing/debugging a knowledge base, the user may at the end of a run encounter a conclusion with 'truth value' UD, however; then it signifies that that conclusion was not needed in the evaluation of the goals.

5.2.3.3. Operators

While many logics are theorem-oriented, the SIMPLEXYS logic is value-oriented. In mathematical terms it is an *algebra* rather than a *logic*. Theorems must be derived through a proof procedure that normally involves a lot of searching; values are obtained through computations by the application of operators, or, preferably, just looked up in operator truth tables. Computationally, the latter is the fastest procedure.

SIMPLEXYS implements the monadic operators *not*, *must* and *poss*, and the dyadic operators *and*, *ucand*, *or*, *ucor* and *alt* (see appendix 2 on additional operators).

Monadic operators

Operators with *1 argument* are *not*, *must*, and *poss*.

v	not	must	poss
TR	FA	TR	FA
FA	TR	FA	FA
PO	PO	FA	TR

```
not  v := if v = TR then FA else
          if v = FA then TR else v

must v := if v = PO then FA else v

poss v := if v = PO then TR else FA
```

Besides *not*[1], we have introduced two more operators: *poss* (poss p: 'no definite value can be found for p'), and *must* (must p: 'p is guaranteed to be true'). It can be shown that with a combination of these three operators all monadic logical operators can be formed. Examples are:

```
v FA PO TR
  ========
  TR FA FA   must not v
  TR FA TR   not poss v
  TR TR FA   not must v
```

Dyadic operators

Operators with *2 arguments* are *and*, *ucand*, *or*, *ucor*, and *alt*. These operators have equal priority; parentheses may be used to force the order of evaluation.

The operators *and* and *or* have a meaning similar to their meaning in two-valued logic.

And / ucand

v \ u	TR	FA	PO
TR	TR	FA	PO
FA	FA	FA	FA
PO	PO	FA	PO

```
u and v := if u = FA or v = FA then FA else
           if u = PO or v = PO then PO else
                                    TR
```

And has the intuitive meaning. The difference between *and* and *ucand* is, that in *u and v*, v is not evaluated when u = FA. In *u ucand v* both u and v are always evaluated. Logically, they are equivalent in that the result of their application is identical.

[1] The fact that NOT PO is defined to be PO can create problems in some cases; these are discussed in section 6.5.2.3.

Or / ucor

v \ u	TR	FA	PO
TR	TR	TR	TR
FA	TR	FA	PO
PO	TR	PO	PO

```
u or v := if u = TR or v = TR then TR else
          if u = PO or v = PO then PO else
                                   FA
```

Or has the intuitive meaning. The difference between *or* and *ucor* is, that in *u or v*, v is not evaluated when u = TR. In *u ucor* v both u and v are always evaluated. Logically, they are equivalent in that the result of their application is identical.

Alt

The operator *alt* is new. In the expression *u alt v*, u and v are logically equivalent alternatives (rules or expressions) that represent different solutions to the same problem. In other logics, it may be possible to have several rules which evaluate to the same result. In Prolog, one can have

```
x :- expr1
x :- expr2
```

This is not allowed in SIMPLEXYS. It violates the convention that all rule names must be unique; no two rules named x are permitted. If expr1 and expr2 are logically equivalent[1], the equivalent SIMPLEXYS construct is

```
x : expr1 alt expr2
```

Such a construct has the added benefit that in the knowledge base all knowledge about x is kept together. The *alt* operator introduces the possibility of a *conflict*; a conflict arises if the two alternatives are discovered to be not logically equivalent, i.e. one of the operands evaluates to TR and the other to FA. This results in the following definition:

v \ u	TR	FA	PO
TR	TR	**	TR
FA	**	FA	FA
PO	TR	FA	PO

```
                   ** = conflict
u alt v :=
          if (u = TR and v = FA) or
             (u = FA and v = TR) then
                                 conflict else
          if (u = TR or v = TR) then TR else
          if (u = FA or v = FA) then FA else
                                     PO;
```

[1] If not, use the *or* operator rather than the *alt*.

Conflicts can arise only if the validation mode (see section 5.5.3) is active, where the evaluation is unconditional, so that both u and v are evaluated. When a conflict is discovered during a logical evaluation in a run of a SIMPLEXYS expert system, the evaluation ideally should be stopped because this is a knowledge base inconsistency error. We let the user decide, however: either consider the error as fatal and stop the expert system's operation immediately, or consider the error to be non-fatal[1] and continue the evaluation, taking as the resulting value that of u; in either case, the inconsistency does result in an error message, however.

Normally, evaluation is conditional, i.e. it stops when u = TR or u = FA, so that no conflict can arise. Only if we cannot reach a definite solution (TR or FA) through u (i.e. u evaluates to PO), we try to reach it through v. Another interpretation is that v is the *default* value for the expression, which is taken only if u evaluates to PO. If more than two alternatives exist, multiple *alt*'s can of course be used, as in: p *alt* q *alt* r *alt* s.

We consider the *alt* operands to be *logically*, but not *semantically* equivalent. Thus, if in the expression *p alt* q, p is found to be either TR or FA, we do *not* give q the same value, nor do we, if q is found to be either TR or FA, give p the same value. Two reasons support this approach, a logical reason and an implementation reason.

The following example clarifies the logical reason. If p, x and y are rules, and p's expression is

```
p : x alt y
```

then, if x evaluates to TR, while y evaluates to (or would evaluate to) PO, e.g. because some knowledge is temporarily unavailable, then setting y to TR might interfere with other rules that rely on y being PO, i.e. on the fact that the knowledge *is* unavailable.

The following example clarifies the implementation reason. If in the rule

```
p : q alt (x or y)
```

q evaluates to TR, there is, in SIMPLEXYS, no way to store the knowledge that *x or y* is TR, because only the *components* of the expression, x and y, can be given values.

History operators

Besides a (logical) conclusion, a rule also has a (numerical) *history counter* which contains the period, in seconds, since the rule's conclusion last changed. History operators do not

---

[1] This is the prudent approach; intercepting this error allows the system to shut down in a graceful way.

perform a logical operation but a *numerical* comparison; they inspect the history counter and can provide answers to questions like 'how long has X been true?', where X is any conclusion.

There are six history operators:

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

History operators are used as follows:

r historyop (*numerical expression*)

where r refers to any rule. The right hand side of the history operator, between '(' and ')', is any valid Pascal expression[1].

Application of a history operator is done in two steps. The result is first set to the value of r (r is evaluated if at this point it was not yet evaluated). If the result is TR or PO, r's history counter value (see also section 5.2.1) is compared, using the history operator, to the value of the numeric expression. If the comparison yields true, the result is unchanged, else the result is made FA. For example, the value of the expression

```
HRN > (120)
```

with interpretation 'the heart rate has been normal for longer than 120 seconds' is given in the following table:

value (HRN)	history (HRN)	value
TR	> 120	TR
TR	<= 120	FA
PO	> 120	PO
PO	<= 120	FA
FA	any	FA

The logic of this table is not free from problems. No problems arise if a rule's conclusion is somehow restricted to the values TR and FA only. Problems do arise if a rule can temporarily have conclusion PO (e.g. due to an invalid measurement) or is marked UD (because the rule is not evaluated in some runs).

Assume that rule HRN in the example above represents the concept 'the heart rate is normal'; that the heart rate has been normal for more than 120 seconds; that 20 seconds ago

---

[1] Note that a history operator is connected to a *rule*, not to an *expression*; an expression has no history counter.

the signal, from which the heart rate is derived, was momentarily not available, leading to conclusion PO (unknown) for HRN at that time instant; and that we want to know whether the heart rate has been normal for more than 120 seconds. Because HRN's history counter will now have the value 20, this situation, depicted in figure 5.3, leads to the counter-intuitive result FA rather than PO.

Figure 5.3. The rule's conclusion may have been true at all times, but at some time it was impossible to establish its conclusion. The history counter reflects the time, indicated by the arrow, where the conclusion last changed. The history operator erroneously returns FA rather than PO.

Correcting this problem would demand somehow storing the values of earlier conclusions. No method could be found to achieve a correct solution in an efficient (time- and storage-wise) manner.

Since there is no way to guarantee a correct history if conclusions can have value PO, only the histories of STATE rules are fully trustworthy. The histories of other rules cannot be trusted, unless the knowledge engineer verifies that during each run the conclusion of a rule whose history is used will be either TR or FA.

An example of a complete rule using another rule's history is:

```
CAN_EXTUBATE: 'extubation is allowed'
ASSIST > (15 * minutes) and RESPIRATION_STABLE
THEN DO writeln ('consider extubation')
```

In this rule, 'minutes' is a Pascal constant with value 60. This rule should be interpreted as:

'*if* the patient has been on ASSIST for more than 15 minutes *and* his RESPIRATION is STABLE *then* extubation is allowed'.

5.2.3.4. THELSEs

There are three types of THELSEs. One, the THELSE GOAL type, is used to specify rules that must be evaluated; this type has an inferencing function, not a logical one. Another type, the THELSE DOs, specifies Pascal code to be executed. The third type, the THELSE TR/FA/PO category, is a collection of implication-like expressions such as 'if q = x then p := y', where p and q refer to rules and x and y are any constant logical value (TR, FA or PO). Their logic is:

```
q : THEN TR p        if q = TR then p := TR
q : THEN FA p        if q = TR then p := FA
q : THEN PO p        if q = TR then p := PO *)
q : ELSE TR p        if q = FA then p := TR
q : ELSE FA p        if q = FA then p := FA
q : ELSE PO p        if q = FA then p := PO *)
q : IFPO TR p        if q = PO then p := TR
q : IFPO FA p        if q = PO then p := FA
q : IFPO PO p        if q = PO then p := PO *)
```

*) At first sight, it seems strange to set a conclusion to value PO. However, sometimes one conclusion implies that *it is known* that another conclusion cannot possibly be derived. This knowledge can be exploited to prevent a later superfluous attempt to evaluate the conclusion.

These expressions state that if rule q has conclusion x, then rule p's conclusion is known to be y; rule p itself need not be evaluated: its conclusion is reached without additional effort.

But this also implies that under certain conditions conclusion p can derived by evaluating rule q. The Inference Engine therefore uses such THELSEs in two directions. An example will clarify this. Given are the following two rules:

```
M : H and L
B : F and E; THEN FA: M
```

The Inference Engine can obtain conclusion M in the following two ways:

- Assume that somehow B is evaluated first. If B evaluates to TR, the conclusion of M is known to be FA. Hence M's expression need not be evaluated[1].
- Assume that somehow M is evaluated first. If the evaluation of M's *expression* leads to conclusion PO (unknown) for M, there is another possibility to reach a more conclusive result for M: evaluate B, because B can give conclusion FA to M. If B subsequently evaluates to TR, M obtains the result FA. If B evaluates to PO or FA, the conclusion of M will continue to be PO[2].

---

[1] In validation mode, however, M's expression is evaluated anyway, in order to detect a possible conflict.
[2] As long as M is PO, the Inference Engine will continue to evaluate rules that can give M the value TR or FA, until there are no more such rules.

5.2.4. ON statements

ON statements describe context switches or state transitions. Each ON statement has the format:

```
ON tg FROM s1 TO s2
```

where

tg is any rule;
s1 is a non-empty STATE rule list;
s2 is a STATE rule list.

Rule tg is called the *trigger rule* or simply the trigger. A state change or context switch takes place if all STATE rules in list s1 have conclusion TR (a STATE with conclusion TR is called an *active* STATE) *and* if tg evaluates to TR. On a change of state, all STATEs in list s1 become FA (inactive) and all STATEs in list s2 become TR (active). List s2 can be empty; an empty list is denoted by the symbol '*'.

When the system starts up, all STATE rules that have an INITIALLY TR (error checking by the Rule Compiler assures that there is at least one) are active. As soon as no STATE rules are active any more, the expert system halts.

5.3. SIMPLEXYS as a programming language

This section describes the structure of SIMPLEXYS 'knowledge programs'. We will show that this structure correlates well with how a real time expert system operates.

5.3.1. A typical operation of a SIMPLEXYS expert system

The following global description of a typical operation of a SIMPLEXYS expert system is from the point of view of the data being processed; the inferencing is extensively described in section 5.5 and appendix 5.

First is an initialization, a start up phase (figure 5.4). The first operation is to *acquire any factual data* (FACT rule conclusions) that may be relevant for the system's operation, fixed knowledge about the current case. Then a *global initialization procedure* (INITG code) is executed, that performs actions such as testing which instruments are connected, initializing and calibrating those instruments, opening disk files etc. This completes the initialization phase.

After the initializations, one or more runs will take place. The knowledge base defines an initial context by means of an INITIALLY TR appended to one or more STATE rules, and

these STATE rules determine by their THEN GOALs which goal rules are to be evaluated in the first run.

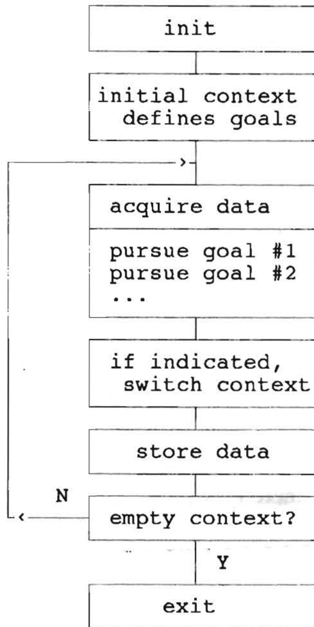


Figure 5.4. A typical expert system operation.

The first action in a run is to *acquire data* or anything else that is necessary for this run; this is done by executing the INITR code. Next all THELSEs are executed and thus the *goal rules are evaluated* and any actions connected to them executed, such as adjustment of instruments, presentation of results, etc. Then the system attempts to perform a *context switch*; this is done by executing the ON statements. After this, a *run exit procedure* is executed, e.g. to display or store the run's numerical results; this is done by executing the EXITR code. Then the expert system program checks whether the new context is empty (i.e. whether any STATE rule still has conclusion TR). If so, another run is started. If not, the *global exit procedure* specified in the EXITG code is executed, e.g. to shut down instruments, close disk files, etc. Then the expert system halts.

5.3.2. SIMPLEXYS programs

A SIMPLEXYS program consists of up to 7 sections. The first 5 program sections are optional; they are necessary only if the rules need to interface with Pascal code (e.g. to perform acquisition of data, print output etc.). Program sections 6 and 7 are mandatory. The program sections need to appear in this order:

1.	DECLS	declarations	
2.	INITG	global initializations	optional,
3.	INITR	run initializations	Pascal code
4.	EXITR	run exit code	
5.	EXITG	global exit code	
6.	RULES	the rules	
7.	PROCESS	the protocol	

5.3.2.1. Declarations

In this program section, all variables, that are used by initializations, exit code, TESTs and DOs, must be declared and their type must be specified; types and variables must be declared in valid Pascal code. Similarly, all procedures and functions that are used in the initializations, exit code, TESTs or DOs must be declared. The Pascal compiler includes the code in this program section into the SIMPLEXYS Inference Engine without changes.

5.3.2.2. Initializations and exit code

Program sections 2 through 5 contain Pascal code. The Pascal compiler includes the code in these section into the SIMPLEXYS Inference Engine without changes.

The statements in the INITG section will be executed when the expert system starts up. The statements in the INITR section will be executed at the start of each run, including the first one. The statements in the EXITR section will be executed at the end of each run, including the last one. The statements in the EXITG section will be executed only once, at the end of the last run.

5.3.2.3. Rule definitions

Program section 6 starts with the keyword RULES. The rules must be inserted here; rules consist of two to four parts:

rule header (name and text string)	mandatory
rule type or rule's expression	mandatory
initial conclusion	optional
THELSEs	optional

1. The rule header. First comes a symbolic name by which the rule and its conclusion can be referenced by other rules. This name consists of alphanumeric characters (numbers are allowed but not recommended) and underline characters (except in first position). Each rule must have a unique name. The name is followed by the character ':', which is followed by a text string that is used in questions, to show results and for explanation purposes. The text may be empty, but it must be surrounded by single or double quotes, and it must be a valid Pascal string.

2. The rule body, which indicates the rule's type. It consists of either:

   a. The special symbol FACT; thus the rule is a FACT rule.

   b. The special symbol ASK; the rule is an ASK rule.

   c. The special symbol TEST; the rule is a TEST rule.
      - The remainder of the line contains text. The text is *one* valid Pascal statement, which has the purpose to assign a value of either TR, FA or PO to the reserved symbolic name TEST; the default value is FA. Example:

        ```
        TEST if HR > 120 then TEST := TR else TEST := FA
        ```

      - The remainder of the line contains no text. The following lines, as many as necessary, consist of one or more Pascal statements. The purpose of the Pascal code is to assign a value of either TR, FA or PO to the reserved symbolic name TEST; the default result is FA. The end of the Pascal code is marked by the word ENDTEST on a new line. Example:

        ```
        TEST
          if HR > 120 then TEST := TR else
          if HR <  60 then TEST := FA else
                           TEST := PO;
          HR_SAVE := HR
        ENDTEST
        ```

   d. The special symbol BTEST (for Boolean TEST); the rule is a TEST rule with a slightly different syntax. The remainder of the line is used to specify a valid Pascal *boolean expression* (not an assignment statement). The Pascal code evaluates to a conclusion of either TR or FA. A multi-line version of BTEST is not implemented. Example:

      ```
      BTEST HR > 120
      ```

      This example is fully equivalent with and only a more compact way to write

      ```
      TEST if HR > 120 then TEST := TR else TEST := FA
      ```

   e. The special symbol MEMO; the rule is a MEMO rule.

   f. The special symbol STATE; the rule is a STATE rule.

g. An expression; the rule is an EVAL (evaluation) rule. The expression cannot be longer than a single line[1].

3. The initial conclusion. The keyword INITIALLY followed by TR, FA or PO; the INITIALLY of STATE rules can be followed by FA or TR only. By default, unless overridden by an INITIALLY, FACT and MEMO rules are initialized to PO[2], and STATE rules to FA. ASK, TEST and EVAL rules are marked UD.

4. THELSEs. Each THELSE line starts with either THEN, ELSE or IFPO; this is followed by one of the keywords TR, FA, PO, DO or GOAL, and then followed by argument(s) or Pascal code.
   a. TR/FA/PO takes as argument(s) other rules, but not FACT and STATE rules.
   b. GOAL takes as argument(s) other rules, but not FACT, MEMO and STATE rules.
   c. DO is followed by one or more Pascal statements.
      - The remainder of the line is not empty. The text following DO is *one* valid Pascal statement. Example:

```
THEN DO writeln ('the heart rate is abnormally low')
```

      - The remainder of the line is empty. Successive lines contain Pascal code, as many lines as needed. The end of the Pascal code is indicated by the word ENDDO on a new line. Example:

```
THEN DO
  writeln ('the heart rate is abnormally low');
  HR_SAVE := HR
ENDDO
```

5.3.2.4. The PROCESS section

Program section 7 starts with the keyword PROCESS. This section describes the dynamics of the process. All state transition or ON statements are inserted here.

The Rule Compiler will recognize the start of this program section by the keyword PROCESS, and the end of the section by the fact that the end of the file is reached.

Frequently, in simple applications where the goals remain the same in every run, the PROCESS section will consist of only one ON statement such as:

```
ON READY FROM RUNNING TO *
```

---

[1] If an expression is longer than one line, split it up into more than one rule; this is easier to understand and almost always more efficient.
[2] PO rather than UD, since UD signifies that a rule's conclusion can be acquired through some type of evaluation.

where RUNNING is the expert system's only STATE rule (and thus it must be INITIALLY TR) which specifies the goal rules, and READY is a rule that tests whether the expert system is ready and must halt. For an example, see figure 5.1.

5.3.2.5. Time keeping

In many final versions of the expert system the time will be obtained from the computer's real time clock. During the design and testing phases, which are normally not real time and where because of testing a run may last any period of time, this is often impractical. While testing, a run which in real time must take 5 seconds might last 5 minutes, upsetting all history counters. Optionally, therefore, a *simulated time* can be specified as a run time option. In this mode it is assumed that runs succeed each other with fixed time intervals, which the user can specify. In most cases, this will make a transition from testing mode to real time mode painless.

Time keeping is done through the variables _time and _time0, according to the following Pascal declaration[1]:

```
var _time, _time0: longint;
```

Their usage is dependent on the type of time keeping:

Simulated time: The user has, as a run time option (see section 5.6.2), specified a 'simulated time' period (in seconds) between successive runs having a constant value _time_inc. This results in inclusion into the Inference Engine of a Pascal declaration like:

```
const _time_inc = 5;
```

On system startup, _time is reset to 0. At the start of each run _time is incremented by _time_inc. Variable _time0 is not used.

Real time: On system startup, _time0 is read from the system clock. At the start of each run, _time is updated (using function sys_time) from the system clock, using the expression

```
_time := sys_time - _time0
```

Thus _time has the meaning: 'the number of seconds elapsed since system startup'. This implies that normally the cycle time of the expert system should be greater than one second and preferably an integer number of seconds.

---

[1] In order to prevent conflicts between names, it is a convention that internally used SIMPLEXYS names start with an underscore character and that the user does not employ such names.

Each rule's history counter is set to the _time at which the rule's conclusion changes, so the Pascal expression

```
_time - _history [rule]
```

has the meaning: 'the number of seconds since the conclusion last changed'; this expression is used when SIMPLEXYS expressions contain history operators. An example:

```
HR_STABLE > (15 * minutes)
```

is translated by the Rule Compiler into the following Pascal code pattern:

```
TEMP := _evalrule[1] (_R [_HR_STABLE]);
if (TEMP = TR) or (TEMP = PO) then
  if not (_time - _history [_R [_HR_STABLE]]) > (15 * minutes)
    then TEMP := FA;
```

where TEMP is a variable of type bool that returns the result of the evaluation of the expression, _R [.] is the array that stores the rules' conclusions, _history [.] is the array that stores the rules' histories, and _HR_STABLE is a constant index, assigned by the Rule Compiler.

Time keeping uses these procedures and functions:

function sys_time: longint; returns the system time (in seconds);

procedure init_time; initializes time-keeping; resets _time; uses sys_time at system start up to initialize _time0 if real time mode is selected;

procedure update_time; updates _time; uses sys_time at the start of each run if real time mode is selected; otherwise increments time by _time_inc.

5.4. Programming in SIMPLEXYS

One of the main characteristics of SIMPLEXYS is that it is designed for real time applications and that it can evaluate its rules with high speed. Common designs of Inference Engines are not efficient: they imply an enormous amount of searching through the knowledge base and in some implementations even a very time consuming 'matching' of consequents before a rule can be applied. The searching and matching in SIMPLEXYS is only done once: by the Rule Compiler (see section 5.6.1). This compiler transforms the separate knowledge chunks into a set of tables (see appendix 4), in which the relations

---

[1] See section 4.4.

between the rules are represented by pointers. At execution time, these tables can be traversed at high speed.

A fundamental principle of SIMPLEXYS is thus, that the knowledge base must be compiled before the knowledge can be used by the Inference Engine. The SIMPLEXYS tools therefore perform two basic operations: compilation of the rules into a correct internal representation, and binding that internal representation of the knowledge with the Inference Engine.

Programming in SIMPLEXYS entails the following steps:

1. Acquire the knowledge to be implemented. This is usually done through interviews with a 'domain expert'.

2. Implement the knowledge using the SIMPLEXYS language. This results in the 'knowledge base' or 'knowledge program'. If the knowledge cannot be formalized because it is unclear, contradictory or incomplete, go back to step 1.

3. Generate the internal format of the knowledge, i.e. the code and tables that the Pascal compiler will include into the Inference Engine. This is done through compilation of the knowledge base by the SIMPLEXYS Rule Compiler. A number of extra passes of the Rule Compiler perform syntactic and semantic correctness checks, both on the 'static' and the 'dynamic' parts of the knowledge base. If implementational errors are signalled, go back to step 2; if contradictions are signalled, go back to step 1.

4. Select any run time debugging options that may be deemed necessary.

5. Compile the expert system, by combining the SIMPLEXYS Inference Engine with the Rule Compiler's output and the run time options. This is done by the Pascal Compiler.

6. Run the expert system.

7. Correct imperfections and errors, repeating steps 1 to 5 until satisfied.

This procedure is called 'rapid prototyping' [Klingler, 1986; Politakis and Weiss, 1984]. It assumes that the major problem is step 1, the acquisition of knowledge, because the domain expert does not know what he knows: much of his knowledge has become subconscious. In the early design stages, therefore, step 6 will yield many imperfections. Such imperfections *are* easy to recognize by the expert, however, and their conscious recognition is an essential step in the acquisition of the knowledge. Step 7 allows the repair of errors that become visible. Because some errors or limitations will only show up in the long run, it is hard to say when the expert system is 'finished'. Logically, never, because it will never be 'complete'; practically, as soon as it is sufficiently 'useful', a very subjective assessment.

5.5. Inferencing in SIMPLEXYS

The purpose of the inferencing process is to derive all necessary conclusions (or *goals*) about data that are offered to the system. Generally, the data will change over time, and therefore the analysis will need to be repeated, possibly many times, and possibly with different goals. Each evaluation of all the goals is called a *run*; a typical run may take 5 seconds, including the acquisition of data and the display of results. Figure 5.5 shows how different goal rules are evaluated in a sequence of runs.

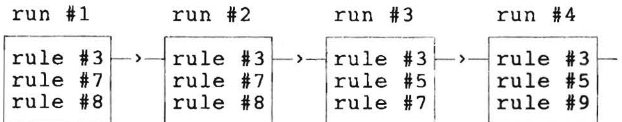


Figure 5.5. A sequence of runs of the expert system and the goal rules to be evaluated in those runs.

The determination of *which* conclusions are to be derived, i.e. which rules are goal rules, is specified by a match between the *prevailing value* of a rule's conclusion at the start of the run and that rule's THELSE GOALs. The arguments of these THELSE GOALs are the *primary goals*, and they are evaluated as follows:

```
for i := first_rule to last_rule do
  case conclusion [i] of
    UD: {do nothing, i.e. do not evaluate the rule!};
    TR: evaluate all THEN GOAL arguments of the rule;
    FA: evaluate all ELSE GOAL arguments of the rule;
    PO: evaluate all IFPO GOAL arguments of the rule;
  end case;
```

The *order* in which the goals are executed thus depends on the ordering of the rules of which they are a goal. The order in which the THELSEs are executed is: first the THELSEs of the ASK rules, then those of the TEST rules, EVAL rules, FACT, MEMO rules, and finally those of the STATE rules[1]. Within a rule type class, the order in which the goals are executed is the order in which the rules appear in the knowledge base text. But in a correct knowledge base this order is immaterial: all goals will eventually be evaluated. What *is* important, is the correct execution order of the goals (or rather of *all* THELSEs) of each single rule. These are executed according to the order which the user specified. In the

[1] To do this efficiently, the Rule Compiler sorts the rules into this order. Note also, that at the start of any run but the first, only FACT, MEMO and STATE rules can already have a conclusion or value (TR, FA or PO); only the conclusions of these rule types are carried over from one run to the next. The conclusions of ASK, TEST and EVAL rules, on the other hand, are undefined (tagged UD) at the start of each run. In the first run, however, ASK, TEST and EVAL rules can have a value as well, if such a value is specified by an INITIALLY.

following example this ordering is important, because data must be acquired before they can be analyzed, and the analysis must precede the presentation of the results (see section 2.3.5 for a similar ordering in VM).

```
X: 'rule X'
STATE
THEN DO acquire_data
THEN TR: DATA_ACQUIRED
THEN GOAL: CHECK_ARTIFACTS, ANALYZE_DATA, SELECT_ALARMS
THEN DO present_alarms
```

Evaluation of a THELSE GOAL argument can cause *secondary goals* to be evaluated, if the argument rule has THELSE GOALs as well. A secondary goal is thus a rule, that is the argument of a THELSE GOAL of a rule that obtains its conclusion during the evaluation of the primary goals.

Normally, the primary goals are attached to STATE rules, because these are meant to describe the problem solving context. They can also be attached to FACT rules and MEMO rules, however. The goals of FACT rules are context independent; their goals will thus be evaluated in every run. MEMO rules can be used to specify goals which are 'condition dependent' rather than context dependent. Goals attached to ASK, TEST and EVAL rules are always secondary goals.

The Inference Engine's sequencing of analyses (shown in figure 5.6) takes care of establishing the context, evaluating the goals that belong to the context and, if necessary, switching to a new context, until no active context exists anymore, i.e. until all STATE rules have conclusion FA.

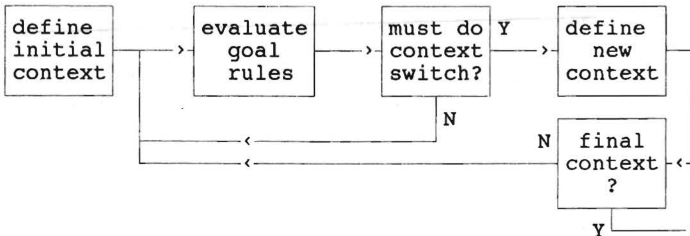


Figure 5.6. The inference loop. At system start up, an initial context is defined, determined by the initially active STATE rules. The goal rules of these STATE rules are then evaluated. The Inference Engine next evaluates the rules which determine whether a context switch must be performed. If so, the new context is defined, and the goals belonging to that context will be evaluated in the next run. If not, the same goals as in the previous run will be evaluated again, presumably with different input data.

The inferencing process that analyzes one 'snapshot' of the data (figure 5.7) takes place within an established context.

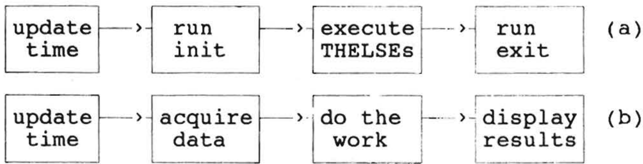


Figure 5.7. Analysis of one data set. Figure (a) gives the general scheme, figure (b) an example. First the current time is acquired. The run initialization code usually obtains the data to be analyzed. Execution of the THELSEs involves the evaluation of the goal rules of the current context. The run exit code can display results or perform other output operations.

In SIMPLEXYS, inferencing is performed in and across a number of *runs*. The inferencing that is performed in one run is called *single run* inferencing, the inferencing that sequences the runs is called *global inferencing*. Each run is a sequence of the following two steps:

1. Given a certain context, evaluate all *primary goals* of that context. Primary goals are usually EVAL rules, although they can also be ASK or TEST rules. As a result, some *secondary goals* may become evaluated as well.

2. Given the context, determine the next context. This step calls for the evaluation of all trigger rules (the rules after the keyword ON in the ON statements), that could possibly cause some STATEs to become FA and others to become TR (frequently some of those trigger rules have already been evaluated in step 1). This step is called a 'context switch'.

In accordance with the above, figure 5.8 shows how in each run not only goal rules are evaluated, but also trigger rules.

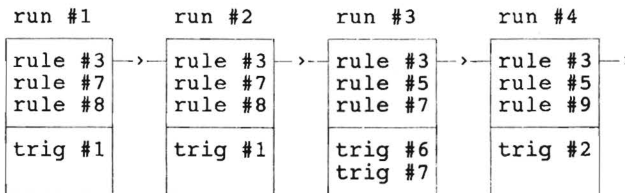


Figure 5.8. A sequence of runs of the expert system and the goal and trigger rules to be evaluated in those runs.

The Inference Engine is both simple and safe. The simplicity is derived from the automaticity of its recursion process. Run time safety is ensured by an abundance of (possibly superfluous) syntactic (and a few semantic) checks. All these tests take very little time and therefore do not degrade the performance.

5.5.1. Single run inferencing

A *single run inference* does the following:

- Update the time.
- Execute the run initialization code, procedure _INITR.
- Execute the matching THELSEs (see section 4.5) of all FACT rules, then those of all MEMO rules and finally those of all STATE rules[1].
- Try to perform a context switch; abort on a deadlock[2].
- Execute the run exit code, procedure _EXITR.
- Undefine all ASK, TEST and EVAL rules for the next run.

Appendix 4 describes the important data structures used by the Inference Engine. Appendix 5 describes the Inference Engine's main procedures and functions.

5.5.2. Global inferencing

The Inference Engine is a Pascal procedure (procedure _infer), which does the following:

1. Initialize the conclusions and history values of all rules to their proper default values (see section 5.3.2.3) or the values assigned by INITIALLYs.
2. Obtain the conclusions of those FACT rules, which did not obtain a value through an INITIALLY, by asking the user. If the FACT rule conclusions are to be obtained in a different way, e.g. by interrogating a data base, the user must provide the appropriate code as a Pascal procedure.
3. Initialize the time.
4. Execute the global initialization code, procedure _INITG.
5. *repeat*
   do a single run inference
   until no STATE rule has conclusion TR anymore.
6. Execute the global exit code, procedure _EXITG.

[1] First run only: start this step by executing the matching THELSEs of all ASK rules, then those of all TEST rules and finally those of all EVAL rules that have an initial conclusion due to an INITIALLY.
[2] A deadlock should not occur, because it is detected and flagged by the Protocol checker; see section 7.3.4.

5.5.3. The validation mode

A validation mode is available for debugging purposes. When debugging and testing an expert system, it is frequently very difficult to provide all combinations of inputs, so that all rules are thoroughly exercized in the attempt to catch as many inconsistencies as possible. The validation mode makes this easier. Essentially *all* rules (within the active context) are fully evaluated.

A selection is possible between two evaluation methods:

1. Conditional or short-cut evaluation. Using this method, in an expression like

   ```
   A and B and C and D
   ```

   A will be evaluated first. If A evaluates to FA, neither B, C nor D will be evaluated. Due to its efficiency, this is the normal evaluation method. It has the drawback that if A is *always* FA, errors due to (conflicts with) rules B, C and D will never be discovered. In general, correctness testing of the knowledge base using this method is quite difficult, as it is very much dependent on the actual inputs supplied.

2. Unconditional evaluation. This method is selected when there is the need to test the consistency of the knowledge base. Now in an expression like

   ```
   A and B and C and D
   ```

   A, B, C and D will all be evaluated, regardless of their values. Due to its inefficiency, this is not the normal evaluation method. Using this method, all questions (from ASK rules), that apply in the current context *will* be asked and all tests (from TEST rules), that apply in the current context *will* be performed.

The validation mode uses unconditional evaluations, but there are other differences, too. Compared to normal inferencing, the following differences are to be noted:

1. The second argument of the operators *and*, *or* and *alt* is evaluated whether necessary or not, i.e. there is no difference between *and* and *ucand*, and between *or* and *ucor*. The resulting *value* will not be different, but the side effects (THELSEs) generally will[1].

2. The second argument of the *alt* operator is evaluated whether necessary or not. A logical contradiction will occur if one operand is TR while the other is FA (recall that *alt* stands for 'logically equivalent alternative'; see section 5.2.3.2). If such an error occurs, the

---

[1] More rules will generally be evaluated, and these may have THELSEs.

system continues[1] and the value of the first operand is assigned to the result of the operation, but an error message is issued. In validation mode the following expression applies for the *alt* evaluation:

```
u alt v := if u = FA and v = TR then FA (but signal error) else
           if u = TR and v = FA then TR (but signal error) else
           if u = FA or  v = FA then FA else
           if u = TR or  v = TR then TR else
                                     PO
```

The expression for u *alt* v given above applies only if the validation mode is active. Normally evaluation is conditional, i.e. it stops when u = TR or u = FA; then no error can be detected.

3. In validation mode a THELSE does not just *assign* a rule's conclusion. Instead, the rule is *evaluated* by function _evalrule and the evaluation's result is compared to the value that should be assigned. Discrepancies are signalled.

4. In function _evalrule, the purpose of step 3 was to obtain a TR or FA conclusion through other rules' THELSEs if 'normal' evaluation did not succeed. Step 3 now is unconditional: the results of *all* THELSEs to the rule are compared with each other and with the evaluated conclusion. Discrepancies are again noted.

In validation mode, lots of superfluous questions may be asked and many extra tests may be performed in order to test for incompatible data and/or inconsistent processing of the data. Therefore this mode is *not* efficient, and should probably not be used in a final design. By default, none of the extra errors is fatal, in order to carry on with the testing, but the user may want to decide to halt the process.

5.6. The SIMPLEXYS Toolbox

SIMPLEXYS is meant to be an easy-to-use toolbox. It provides a number of instruments (tools) for the implementation of expert systems:

1. The SIMPLEXYS language has already been discussed extensively.
2. The SIMPLEXYS Rule Compiler (see section 5.6.1) translates the knowledge into an internal representation that can be easily checked (by the Semantics Checker and the Protocol Checker) and managed (by the Inference Engine).
3. The Semantics Checker (see chapter 6) is an additional pass of the Rule Compiler, which performs a number of semantic correctness checks on the static component of the knowledge base.

[1] But the user may intercept the error and halt the system.

4. The Protocol Checker (see chapter 7) is an additional pass of the Rule Compiler, which performs a number of semantic correctness checks on the dynamic component of the knowledge base, the protocol.
5. The Options Builder (see section 5.6.2) is a small program that can instruct the Inference Engine to select certain debugging options.
6. The Inference Engine (see section 5.6.3), whose mechanism has been described in section 5.5, implements the necessary 'reasoning' ability. By combining the internal representation of the knowledge generated by the Rule Compiler with the Inference Engine, a ready to run expert system results.
7. The Debugger/Tracer (see section 5.6.4) is a tool to examine the inferencing process of the expert system while it processes symbolic information.

5.6.1. The SIMPLEXYS Rule Compiler

The Rule Compiler is very simple. It is of recursive descent type [see e.g. Wirth, 1976b] , where the 'descent' is defined by the SIMPLEXYS syntax.

The SIMPLEXYS Rule Compiler expects an input file containing a 'knowledge program' in the syntax described above; the file extension must be '.rul'. This 'knowledge program' is created and edited using any editor or ASCII-output word processor. The knowledge program source code is then compiled into Pascal code, resulting in several output files that contain both tables (the internal representations of both static and dynamic nets, stored as arrays; see appendix 4) and executable code (for the INITG, INITR, EXITR and EXITG sections and the Pascal code of the TEST rules, history expressions and DOs). The Rule Compiler checks the rules for all syntactic and some simple to detect semantic errors, and it checks the completeness of the rule set; it checks the non-Pascal text only. It halts at the first error it finds and shows the offending line with a self-explanatory error message.

File *rdecl.qqq* will contain the user's declarations from the DECLS section, the code from the INITG, INITR, EXITR and EXITG sections (in the procedures _INITG, _INITR, _EXITR and _EXITG), the code of TESTs (in the function _FTEST), the code of all history checks (in the function _FHISTORY), and the code of all DOs (in the procedure _FDOS).

File *rinfo.qqq* will contain all tables, either as Pascal const or var array declarations (see appendix 4).

The Rule Compiler error checks are mostly simple. Some are:

1. There are no rules.
2. Duplicate rule name.
3. There are no STATE rules.
4. No STATE rule is INITIALLY TR.
5. There are no ON statements.

6. A FROM or TO list contains a non-STATE rule.
7. A rule is unconnected (in no way used by other rules).

The most complex error check is:

8. Incomplete rule set. Often, this is caused by a typing error: a rule needs for its evaluation another rule which cannot be found because its name is misspelt. Another possibility is that a rule is *referenced* (in an evaluation expression, by a THELSE or in an ON statement), but not *defined.*
   In the design stages, this 'forgetting' to define a rule can be done to advantage. The Rule Compiler can, optionally, generate missing rules automatically; they will become rules of type ASK. This 'forgetting' allows knowledge acquisition and implementation to proceed in an orderly, top-down manner, if design is started with the top level rules only. Since no rule text string is available for missing rules, it will be copied from the rule name, so that the rule name itself will be used as the question prompt and explanation text.

In addition, the Rule Compiler checks for a number of internal errors, e.g. 'rule store overflow', which may occur due to a too large number of rules in the knowledge base. The maximum number of rules that can be compiled is currently, due to the limited memory size of MS-DOS machines, approximately 400 to 600, depending on the complexity of the rules.

The Rule Compiler output consists of a number of text files containing arrays and Pascal code sections, that are to be included into the bare Inference Engine to yield a complete expert system.

The DECLS section will be included as such, directly after SIMPLEXYS's own type definitions and declarations of internal variables and procedures. The Rule Compiler does not check any of the Pascal code in this section; that is left to the Pascal compiler.

The code in the INITG, INITR, EXITR and EXITG sections will be gathered into the procedures _INITG, _INITR, _EXITR and _EXITG. The code in the INITG section, for example, is packaged into a procedure as follows:

```
procedure _INITG;
begin
  {insert INITG code here}
end;
```

Such code will be called at the appropriate time by the Inference Engine.

The code in the RULES and PROCESS sections is translated and tokenized into SIMPLEXYS tables and arrays (see appendix 4), that will be used by the Inference Engine. State transitions are tokenized and stored into a table that is traversed by the Inference Engine when it must perform a context switch.

The translation of FACT, ASK, MEMO and STATE rules is trivial; they have no arguments so only their type needs to be stored. The Pascal code of all TEST and BTEST rules are gathered into a procedure _FTEST, which is a large case-statement, where the case index is the rule's sequence number:

```
function _FTEST (t: word): bool;
var TEST: bool;
begin
  TEST := FA;
  case t of
   {TEST statements or bodies are inserted here}
   ...
   {rule 5: BTEST x = 0}
   5: begin
        if x = 0 then TEST := TR
      end;
   ...
   {rule 7: BTEST x > y + z}
   7: begin
        if x > y + z then TEST := TR
      end;
   ...
  else {no valid case}
    fatal_error ('invalid TEST # ', t)
    {this internal inferencing error should never occur}
  end; {case}
  FTEST := TEST
end;
```

Evaluation rules are tokenized and stored into a table, that is traversed by the Inference Engine when it must evaluate a rule. For each evaluation rule, an index is created into a large array, where all the expressions are stored in prefix notation, as tokens (for the keywords) and rule numbers[1]. An example: the expression

```
(F AND H) OR (J AND K)
```

is stored as

```
OR AND F H AND J K
```

In addition, the Pascal code for history tests is compiled into a function _FHISTORY, just like the Pascal code of the TEST rules is compiled into function _FTEST.

Code for THELSEs is stored into tables, similarly to the expressions. The Pascal code for THELSE DOs is stored into a procedure _FDOS, the same way as TESTs are stored; DO code sections are copied without change and without checks.

---

[1] Both prefix and postfix notation avoid the storage of parentheses. Prefix notation was chosen because it allows a simple implementation of conditional evaluations.

As mentioned before, missing rules can automatically be generated; they then become ASK rules. This is easy to do. The Rule Compiler inserts all rule names into a table as soon as they are encountered, and it also keeps track of which rules have been defined. If, at the end of compilation, the table still contains undefined entries, these can easily be added as ASK rules, which do not need arguments (the string is copied from the rule's symbolic name). Using this option, a missing rule in a TO or FROM list will thus become an ASK rule; this will subsequently lead to the error 'FROM or TO list contains a non-STATE rule'.

The Semantics Checker and the Protocol Checker run without user interaction. They act as extra passes of the Rule Compiler that perform static and dynamic semantic checks; these checks are expanded upon in chapter 6 and 7.

5.6.2. The SIMPLEXYS Options Builder

The Options Builder is a program that asks for a number of run time options and then builds a file 'options.qqq'. The options are used in the Inference Engine for these purposes:

1. Real time or simulated time. During the design and testing phases it is necessary to mimic the final system as closely as possible. The major problem is usually the real time nature of the final system. We expect that most frequently a final system will use a run to analyze data that are acquired at regularly spaced time intervals and take actions as a result. When using simulated time, the user can specify the time period, in seconds, that the expert system should assume to occur between successive runs. Simulated time then creates rule histories as they would exist in the final system.

2. Debugging mode. During the design and testing phases it is often to advantage to be able to see the intermediate results of the inferencing process. It is possible to choose sub-options such that more and more details become visible on the computer's screen. These sub-options exist:
   a. show the conclusion of a rule as soon as the rule obtains a conclusion;
   b. show the progress of the inferencing mechanism by messages like 'starting the evaluation of rule ...', 'finished the THELSEs of rule ...' and 'obtained the conclusion of rule ...';
   c. show operators when they are applied, as well as the result (value) of their application;
   d. show the table positions which correspond with the operation the Inference Engine is currently executing;
   e. show, after each run, all conclusions that have been derived, i.e. all rules with a conclusion TR, FA or PO, as well as their previous conclusion and their history count;
   f. use validation mode (see section 5.5.3).

3. Dump results. During the design and testing phases it is often to advantage to be able to investigate the final results of the inferencing process at leisure. All results, i.e. user

interaction and/or debugging information, can be stored either to a printer or to a disk file.

5.6.3. The SIMPLEXYS Inference Engine

The SIMPLEXYS Inference Engine is entirely written in Pascal. The expert system code consists of the coded inferencing mechanism, described in section 5.5 and appendix 5, and *include* compiler directives[1] for the Rule Compiler's output files. The Pascal compiler checks for any errors in the knowledge program's Pascal code sections.

The Inference Engine is a Pascal procedure. The main program, which prints a header text and then invokes the Inference Engine, is short, simple and self-documenting. It may readily be modified for other applications. The SIMPLEXYS program can, as a procedure, be part of another program.

To give an impression of the size of the executable programs: the compiled size of the Inference Engine itself is about 30 Kbytes. The size of 'knowledge programs' depends very much on the number of rules and the size of the Pascal sections that they contain. A small expert system would be about 35 Kbytes, a medium-sized one 80 Kbytes and a large one 300 Kbytes. The blood pressure controller of chapter 9 is a medium-sized application; its size is 112 Kbytes.

5.6.4. The SIMPLEXYS Debugger/Tracer

The design of a correct knowledge base is difficult, and in this design process, testing and debugging are the most difficult [Pau, 1987; Hendler, 1988]. In comparison with systems written in a procedural language, rule based systems are extremely hard to debug. For an review of debugging problems and for more details on the SIMPLEXYS Debugger/Tracer see de Hair [1988] and Philippens [1989].

One difficulty in debugging a knowledge base is to discover that a problem or shortcoming still exists; success in debugging is possible only if the knowledge engineer is a near-expert in the problem domain. Another difficulty is how to locate a problem. Despite a careful design process, incorrect actions may still be observed during the expert system's performance. The best method to locate an error in a knowledge base is to follow this 'hypothesize and test' algorithm:

1. Discover and describe the symptoms.
2. Guess which error might cause these symptoms.
3. Guess which additional symptoms the error would cause.

---

[1] This is because Turbo Pascal does not offer separate compilation and linking of program modules.

4. Discover whether these additional symptoms actually occur; if not, go back to step 2.
5. Locate the moment when the symptoms first occur, and trace back to the cause.
6. Repeat and refine the process until the error is found.
7. Determine how to repair the problem.
8. Repair the problem.

This debugging process will be much easier and faster if debugging tools are available, particularly a tool that allows tracing (step 5 above). Debugging aids are like stethoscopes, necessary to isolate the cause and location of an error. Three different categories of debugging tools can be distinguished, all of which are available in SIMPLEXYS.

Snapshot tools give a picture of the program or the program's variables at a certain point in time. Two kinds of snapshot tools exist: program listings (before execution of the program) and specific/non specific dumps (during the execution of the program). Listings of SIMPLEXYS programs are well structured and easy to peruse. During a run of a SIMPLEXYS program, a dump of all currently evaluated conclusions can be generated at any point in the inferencing process by calling procedure dump, a utility (see appendix 3), e.g. by appending a 'THEN DO dump' to a rule. A same dump can be obtained by answering an ASK rule's question with the symbol '!'.

Dynamic tools show the program or its variables in operation. We distinguish two different kinds of dynamic tools:

- *Tracers* indicate which statements are executed and in which order. In debugging, there are two dimensions to be traced: space and time. The space dimension refers to the storage space of the program, and the time dimension refers to the completion of computation cycles during the execution of the program. Tracing the time dimension is usually most important and most time consuming, but debugging aids should allow the programmer to trace both dimensions.

- *Variable displays* show, while the program is running, the value of one or more variables as they change.

The SIMPLEXYS Inference Engine supports several levels of dynamic displays, selectable by the Options Builder (see section 5.6.2).

Interactive tools offer the user broad powers to stop the execution of the program in arbitrary places and under broadly specifiable conditions. During such suspensions of execution, these systems allow the user to examine such internal status information as the values of variables. These systems enable the user to study error phenomena in minute detail, and they support the process of unravelling the causal threads leading to the first manifestation of an error which might have actually occurred long before.

We distinguish between three kinds of interactive tools: systems which offer the user to suspend execution (1) at any specified program location (the breakpoint capability), (2) when any specified program variable changes value (the watchpoint capability), or (3) after some fixed prespecified number of program statements have been executed (the program stepping capability) [Osterweil, 1983]. The SIMPLEXYS Debugger/Tracer is such an interactive tool (see figure 5.9). It can be used in two ways.

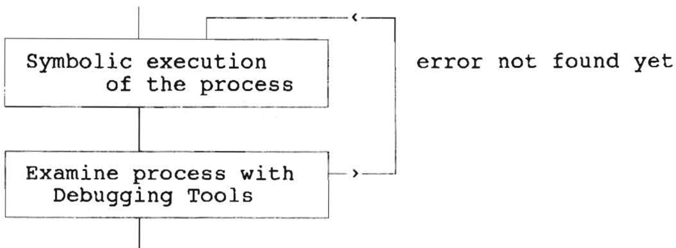


Figure 5.9. The interactive debugging process.

First, during simulations where the input data is not of a time-critical nature, it can analyze the Inference Engine's inferencing steps *while* the expert system is running. The user is able to 'step out' of the inferencing process into an analysis process to ask questions like '*why* is this rule evaluated' and '*how* did this rule get its conclusion'.

Second, and much more important, it can, *after* a real time expert system has finished its operation, analyze the debugging information that the expert system has stored to disk. In this latter mode, the debugger/tracer examines *symbolic* information only; by discarding all questions to the user and all Pascal code, it completely mimics the expert system at the symbolic level. To understand its operation, we have to describe how storing as small as possible a quantity of information to disk (to file *simplex.sav*) during the actual operation of the expert system will allow us to analyze how it progressed through its inferencing steps.

During tracing/debugging, the Inference Engine considers *symbolic* data (rules and their conclusions) only, *not* the user's Pascal code. Thus there is no user-defined interface to the outside world, which means that the Tracer/Debugger can freely use keyboard and screen; this means, moreover, that tracing/debugging is in no way time-critical.

All the user's Pascal code must thus be eliminated, but without effects on the inferencing process. Recall that the Inference Engine's interface with the outside world is through FACT and ASK rules, through the Pascal code in INIT and EXIT sections, TEST rules, DOs and history tests, and through the system's time keeping. We will describe how this information can either be discarded or stored to disk in a compact format (for details, see Philippens [1989]).

1. The Pascal code of the INIT and EXIT sections and DOs can be discarded without influencing the symbolic processing.

2. When the expert system starts up, the INITIALLYs determine the Inference Engine's initial conditions. Then the conclusions of the FACT rules are acquired; these can be viewed as initial conditions as well. Thus the set of all rules' conclusions *after* the acquisition of the FACT rules' conclusions characterizes the 'extended' initial conditions. These are stored to disk.
3. The start time of every run must be stored to disk.
4. The conclusions of all ASK and TEST rules represent the questions asked and the tests performed. These are stored to disk as soon as they are evaluated.
5. The history expressions can contain any Pascal code, but the application of a history operator results in just a boolean value. This value is stored to disk as soon as it is evaluated.

The complete *symbolic* operation of the Inference Engine can be replayed by using, besides the knowledge base itself, only these stored data. Numerical input data are of course lost, as well as all generated output.

The storage of data is very compact; the time takes 32 bits per run, and each rule's conclusion can be stored in two bits only.

The debugger/tracer is thus identical with the expert system with these exceptions:

1. All Pascal code is disregarded; the processing is purely symbolic.
2. At start up, the 'extended' set of initial conclusions, i.e. including the FACT rule conclusions, is recovered from disk.
3. Instead of employing the system time, the stored time is recovered from disk at the start of every run.
4. Whenever an ASK or TEST rule's conclusion or the result of a history operator is needed, the value is recovered from disk.

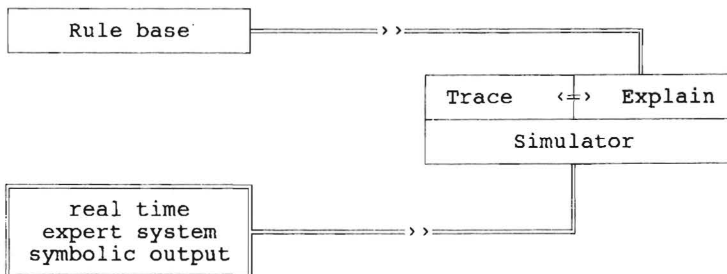


Figure 5.10. The process of debugging the rule base.

The debugger/tracer has two modes, *trace* and *explain*, as shown in figure 5.10. The user can switch back and forth between the two.

In *trace* mode, the real time system's stored symbolic output can be analyzed; there are options like *single step*, *run until* or *go back to* a certain time, *run until* a specified rule gets a specified conclusion, *run until* a specified rule gets any new conclusion, etc. A time trace of the conclusions of a selectable number of rules can be shown in a graphical format in which it is easy to recognize patterns and correlations; an example is shown in figure 5.11.

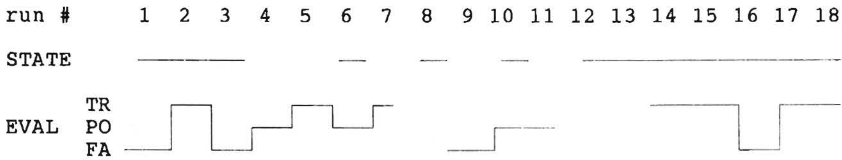


Figure 5.11. Display of traces of a STATE rule and an EVAL rule conclusions. STATE rule conclusions are either TR or FA. The drawn line represents the value TR; when the value is FA, no line is drawn. This way only one display line is necessary to display the trace. EVAL rule conclusions can be TR, FA or PO. These values are all drawn at different levels by a continuous line. Three display lines are now necessary to display the trace.

Cursor keys allow moves through the trace. Moving to the right or to the left selects the next or the previous run. Moving up or down selects the 'active rule', whose information (name, text string, type, initial value etc.) is displayed in a text window.

In *explain* mode the knowledge base needs to be consulted as well; there are options like *show the current goals*, *show the goal* that is currently pursued, *show the rules* that have given the current rule its conclusion, etc. The 'why', 'how' and 'when' questions are the most important:

- '*Why* is it important to determine that ...'. Asking *why* questions is moving into the direction of the root or goal in the rule's evaluation tree.
- '*How* was it established that...'. Asking *how* questions is moving toward the leaves or primitive rules of the rule's evaluation tree.
- '*When* was it established that...'. In contrast with the other two types of questions, we do not move in the evaluation tree but ask for the history counter value of that rule.

Several built-in explanation systems for different expert systems all try to answer these questions [Coombs, 1984; Gupta and Prasad, 1988; Khoroshevsky, 1985], but the implementation and layout of these systems heavily depend on the type of expert system to be elucidated and the degree of knowledge of the user. We, too, found that the explanation must be in terms of well-known SIMPLEXYS concepts. To answer the above type of questions, the information and the relevant section of the knowledge base are shown graphically.

In the following examples, we use a rule's sequence number rather than its name. Numbers preceded by S indicate a STATE rule, numbers preceded by R indicate any rule. The tracer/debugger uses numbers as well; names are often too long to allow a convenient display. The correlation between name and sequence number can be found in file *rinfo.qqq*, generated by the Rule Compiler.

An example of the display of an evaluation rule 'tree' is shown in figure 5.12. The rule's knowledge base text is:

```
R1: '...'
(NOT R2 AND (R3 OR R4 OR R5 OR R6)) OR R7
THEN FA: R10, R11
THEN TR: R12, R13
```

Besides, rule R8 has a THEN FA to the rule.

The display does not only show how the rule is connected in the net (see chapter 6), but also the current values of the relevant conclusions. Note that rules R5 and R6 have thus far not been evaluated.

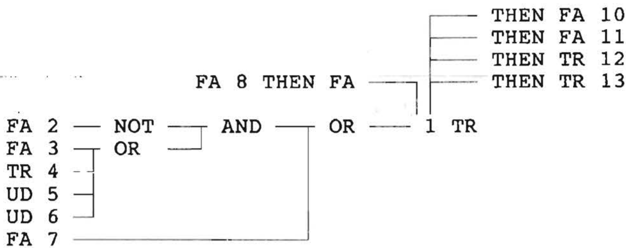


Figure 5.12. An evaluation rule display.

Figure 5.13 displays an ON statement where the trigger rule is the display's center. Figure 5.14 shows a part of the protocol; the STATE rule is the now display's center, and all trigger rules leading to it and leaving it are shown.

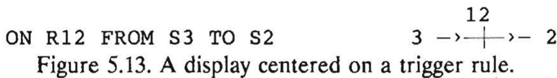


Figure 5.13. A display centered on a trigger rule.

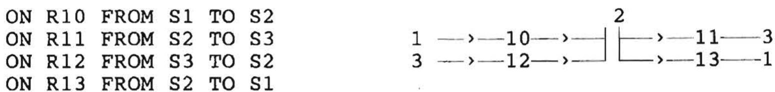


Figure 5.14. A display centered on a STATE rule.

The display is not large enough to show all rules and their connections at the same time, but there is usually room for several of the above displays at the same time; the rules

displayed and the display format can be selected by the user. Moreover, cursor keys conveniently allow movements through the net and the protocol, in order to select a new display centered on any of the neighboring rules shown on the display.

5.7. Worst case performance of SIMPLEXYS expert systems

In this section we will demonstrate that the worst case time required to evaluate all goals in a SIMPLEXYS knowledge base is approximately linear with the total number of rules in the knowledge base. We will not offer a proof, but an informal demonstration why this should be so.

In practice, each context often requires a fair percentage of all rules to be evaluated. The worst case upper bound for the time required to evaluate all rules in a certain context is obtained when *all* rules must be evaluated. A rule is either a primitive rule or a composite (evaluation) rule. Primitive rules can be evaluated directly; their evaluation does not depend on other rules. Thus the time required for the evaluation of a primitive rule can be acquired by actually evaluating it. Composite rules reference other rules; to evaluate a composite rule requires inferencing, evaluation of other rules and operators.

In this discussion about timing we will not consider ASK rules, because the time taken to answer a question is not under the control of the expert system. Thus, if the rule is a primitive, it can be either a FACT, a TEST, a MEMO or a STATE rule. Evaluation of a FACT, MEMO or STATE rule is just recovering its previously stored value. This recovery is a simple process, which requires a constant measure of time T_p. Let the combined number of primitive rules of type FACT, MEMO and STATE be N_1. The total time required to evaluate all these rules is thus

$$T_1 = N_1 * T_p.$$

If the rule is a TEST, we assume that the evaluation time required to evaluate it is finite and constant. If it is not finite, we are in serious trouble because no tools exist to check the Pascal code for finiteness, but usually insight into the code combined with testing of each single TEST rule can provide a high level of confidence. The required evaluation time is not the same for all TEST rules; let the *average* time to evaluate a TEST rule be T_t. If there are N_2 TEST rules, the time to evaluate them is

$$T_2 = N_2 * T_t.$$

Evaluation of a composite rule calls for a) evaluation of a number of 'subordinate' rules, and b) evaluation of the expression. If the subordinate rules are primitives, the time necessary to evaluate them has been accounted for in T_1 or T_2, and the extra time required is due to the application of the operators. If one or more of the subordinate rules is composite as well, evaluation of the rule again calls for a) evaluation of a number of

'subordinate' rules, and b) evaluation of the expression. The time taken to evaluate *all* composite rules is thus related to the total number of operators in the knowledge base. Let there be N_3 composite rules with a total of N_o operators, let the time to evaluate one operator be To, and let the overhead time to evaluate a composite rule be T_c. Evaluation of all composite rules thus takes a time

$$T_3 = N_3 * T_c + N_o * T_o$$

excluding the time necessary to evaluate all primitive rules that are referenced by those rules.

Thus far we have not considered the THELSEs that may be executed. Worst case is that a THELSE TR/FA/PO to a TEST rule has no effect, because the rule that it is directed at has already been evaluated. This is due to the fact that application of a THELSE is generally faster than an evaluation. A THELSE TR/FA/PO to a MEMO rule only takes a fixed overhead time. Worst case for a THELSE GOAL is evaluation of its goal rules, but this has already been accounted for in T_1, T_2 and T_3. Worst case for a THELSE DO is execution of its DO code. Let there be a total of N_4 THELSEs, let the overhead time of evaluating a single THELSE be T_h, and let the *combined* time to execute all DOs be T_d. Executing all THELSEs then takes

$$T_4 = N_t * T_h + T_d.$$

This leads to a total worst case run execution time of

$$T = T_1 + T_2 + T_3 + T_4 = \\ N_1 * T_p + N_2 * T_t + N_3 * T_c + N_o * T_o + N_t * T_h + T_d$$

which is linear in the number of rules. To this should be added a context switch overhead, which mainly depends on the complexity of the protocol (but note that the time required to evaluate the trigger rules has already been accounted for). T_p, T_c, T_o and T_h are processor-dependent but can be measured. T_t and T_d can be measured or estimated. The numbers N_1, N_2, N_3, N_o and N_t can be obtained from the knowledge base. In practice, the proportions of these numbers are often relatively constant.

The above is a simplification. In order to obtain more insight into the evaluation times required, some simulations were performed. The following tests were executed on an IBM-XT compatible PC with a 10 MHz 8088 processor. The tests consisted of evaluations using the following rule sets:

test 1: evaluation rules, no operators, recursion

```
rule 1 : BTEST true;
rule i : rule i-1, 1 < i < N
GOAL   : rule N
```

test 2: evaluation rules, one operator, recursion

```
rule 1 : BTEST true;
rule 2 : rule 1 and rule 1
rule i : rule i-1 and rule i-2, 2 < i < N
GOAL   : rule N
```

test 3: primitive rules only, no recursion

```
rule i : BTEST true, i > 0
GOAL   : rule 1, rule 2, ..., rule N
```

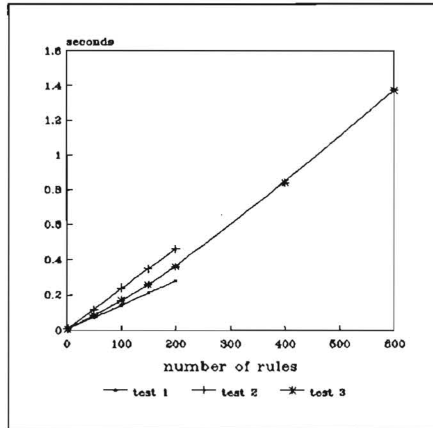


Figure 5.15. Test run times on a 10 MHz 8088 computer.

The run times of these tests (with a very simple protocol) are shown in figure 5.15; results for tests 1 and 2 for a nesting of more than 200 rules cannot be obtained due to an overflow of the processor's stack[1]. The tests indicate an inferencing overhead (context switch and bookkeeping) of approximately 8 ms per run, a recursion time of about 1.4 ms per evaluation rule without operator, about 1.0 ms for the evaluation of one operator plus one rule's conclusion lookup, and about 1.7 ms for the evaluation of one THELSE plus one simple primitive rule. As a global rule of thumb *for this 10 MHz 8088 processor* we provide these times as rounded estimates:

overhead per run	8 ms
T_p = evaluation of a FACT, MEMO or STATE rule	1 ms
T_t = evaluation of a simple TEST rule	1 ms
T_c = evaluation of an EVAL rule	1 ms
T_o = evaluation of an operator	1 ms
T_h = evaluation of a THELSE	1 ms

[1] In practical cases, this will generally not be a limitation.

With a faster processor, these times decrease proportionally.

5.8. Summary of efficiency issues

To summarize, the efficiency of SIMPLEXYS is most of all due to these factors:

1. Rules are evaluated only once (each run). Thus each node of the problem network is visited once at most, preventing a combinatorial explosion of the search space. Evaluation takes linear time.
2. Rules can yield multiple conclusions from one evaluation.
3. Rules are evaluated only in those contexts in which they are relevant. The protocol makes it possible to specify which rules need to be evaluated in which context. This effectively eliminates the evaluation of inappropriate rules.
4. Links between rules are compiled; there is no searching for the rule(s) to be evaluated next.
5. The type of logic implemented allows fast evaluation using truth tables, not complex formulae.
6. The operators provide conditional evaluation; rules are not evaluated if their conclusion is irrelevant.
7. Data are stored in known memory locations, not in lists; there is no searching for data.
8. The Inference Engine is efficient. It is implemented in Pascal, a very efficient language compared to LISP and PROLOG, the usual expert systems implementation languages[1].
9. The full power of Pascal is available to the knowledge base designer. Pascal code can be included into the rules, e.g. to perform calculations, to acquire new data or to implement a user interface. Essentially, the SIMPLEXYS language is a superset of Pascal[2].

---

[1] The efficiency can be increased by about 50% by compiling the Inference Engine code with some non-default compile options: no stack checking, no array index checking, etc. This is not recommended. If greater efficiency is wanted, use the C version of SIMPLEXYS. It is 3 to 4 times faster.

[2] For the SIMPLEXYS C version, replace 'Pascal' by 'C'.

6. Checking the semantics

This chapter discusses the theory on which the SIMPLEXYS Semantics Checker is based; for more information and some additional background see Boon [1987]. It describes how the basic elements of SIMPLEXYS, the rules, are linked together into what is called a *net*, how evaluation takes place and how errors can be found. Errors occur if a rule's conclusion somehow could obtain two different, and thus conflicting, values; both evaluations and THELSEs must be considered, since these lead to conclusions, as well as the ways in which they interact.

In the examples in this chapter a simplified rule syntax notation is used. Since we consider evaluations, THELSEs and their interactions only, we will use a shorthand notation for each of these single elementary parts. Also, a rule name beginning with an E is an evaluation rule, a rule name beginning with a P is a primitive rule, and a rule name beginning with an R is any rule. For example, from the evaluation rule definition

```
E1: 'this is rule E1'
E2 AND E3 OR (P1 AND P2) OR P3
THEN TR: E5
THEN FA: P7
```

we derive the following shorthand notation for its three elementary parts:

```
E1: E2 AND E3 OR (P1 AND P2) OR P3          the expression
E1: THEN TR E5                              first THELSE
E1: THEN FA P7                              second THELSE
```

Evaluation rules play a special role in the net because only they have an expression.

6.1. The need for knowledge acquisition support

The development of expert systems is difficult and time consuming. According to Waterman [1986], it takes five to ten person-years to build an expert system that can solve a moderately complex problem. Early systems such as MACSYMA and DENDRAL took much more effort to build, each over 30 person-years. This is due not only to the difficult nature of the task but also to the lack of sophisticated and refined knowledge engineering tools to assist in the acquisition, documentation and debugging of knowledge. PROSPECTOR, for example, was directly implemented in INTERLISP, a powerful but rather low level language as far as expert system building tools go[1].

---

[1] Building a new application is of course much easier if a previous application has resulted in a set of tools that can be reused.

The acquisition of the domain knowledge, i.e. the process of extracting the knowledge from a human expert, is by far the most difficult part of building an expert system, because the human expert has no ready access to his own store of knowledge. Support in this area is most needed. One of the most wanted forms of support is generally considered to be a program to check the correctness of the knowledge as it is implemented in the expert system: a knowledge base checker/debugger. One such tool has already been described in section 5.6.4, but that tool analyzes an expert system's *run time* performance. Testing the integrity of the knowledge base at *compile time* is even more important since it *prevents* run time errors from occurring; errors are discovered in an earlier phase of the design process.

It is very important, but also very difficult, to ensure that the expert system will give correct solutions to the problems it is expert at. In the process of verifying that a system is accurate and reliable, the major task is checking that the knowledge base is correct and complete [Suwa et al, 1984]. Unfortunately, no formal mathematical methods are available to analyze the system's deep knowledge; analyzers do not *understand*. Knowledge base debugging therefore usually involves repeatedly testing and refining the system's knowledge. This is necessary because many errors can arise in the process of transferring expertise from a human expert to a computer system. Even if we assume that the expert is perfect, three types of errors exist. The knowledge base can contain errors or limitations because:

- the expert's knowledge is incompletely transferred;
- the knowledge engineer's interpretation of the expert's knowledge is erroneous;
- the implementation (transcription, instrumentation) of the knowledge into the support language's syntax and semantics is erroneous.

The standard way of debugging the knowledge base is by observing the system's behavior in a great number of test cases. Although this is an essential part of testing the consistency and completeness of a knowledge base, no amount of testing can ever guarantee that the knowledge base is completely correct [Dijkstra, 1972]. Especially for expert systems that have to operate autonomously and without human supervision and interaction, it is an unpleasant idea that the knowledge base may contain hidden errors. One of the SIMPLEXYS applications is a clinical 'smart alarms' expert system for monitoring the integrity of a patient's artificial respiration during anesthesia [van der Aa, 1990], and another, an automatic blood pressure controller, is described in chapter 9. It is obvious that such systems should be as reliable as possible.

A better way to strive for reliability is to try to design a program that checks the knowledge base during all the system's top-down development stages and thus maintains completeness and consistency of the knowledge base as it expands. Some work in this field has been reported, e.g. Nguyen [1988] and TEIRESIAS [Davis, 1976], a program that provides aids for debugging an EMYCIN knowledge base. Step-wise development of the knowledge base is usually a good idea; catching errors is almost always easier in the early development stages.

No program can offer full-proof checking, because no program has a deep understanding of the domain knowledge. But some aspects of the knowledge base can be systematically checked by algorithms: its *logical* (syntactic, and partially semantic) completeness and consistency. Static nets encode the static aspects of the knowledge base and are discussed in this chapter. Protocols encode the dynamic aspects of the knowledge base and are discussed in the next chapter. Both are ideally suited for such checks.

6.2. Semantic nets: nodes and links

Although to the knowledge base developer SIMPLEXYS expert systems appear to be rule based, internally the knowledge is represented as a net. Quillian [1967] developed the idea of the *semantic net*, consisting of *nodes* to represent concepts and associative *links* between those nodes to represent meaning in some way[1]. His nodes were dictionary entries, and the 'meaning' of such a node was the collection of nodes that could be reached through one or more links. With semantic nets, Quillian could introduce notions like the 'nearness' of concepts in terms of the number of links between concepts, and he could answer questions like 'what is the similarity between a car and a bicycle', in net terms: 'which nearest nodes are common to the concepts car and bicycle', and 'what is the difference between a tiger and a lion', in net terms: 'which nearest nodes are not common to the concepts tiger and lion'.

For most other types of questions such a definition of 'meaning' is unmanageable. The 'first order meaning' of a concept (node) is the collection of all concepts (nodes) pointed to by links originating in the initial concept, i.e. the dictionary explanation (or rather expansion) of the concept. The equivalent, but further expànded 'second order meaning' is the usually much larger collection of all concepts pointed to by two successive links, i.e. the collection of all dictionary expansions of the dictionary expansion of the concept; and so on in ever widening circles. The ultimate 'meaning' of a concept would probably include most dictionary entries. Such definitions are essentially circular.

Semantic nets became much more manageable, when 'primitive' nodes were introduced, nodes which themselves are obvious in some sense and need not be defined in terms of other nodes. Primitives seem to be possible in 'micro-worlds' only, such as Winograd's [1972] blocks world, where the blocks and the operations on them are primitives. This is not much of a limitation in expert systems, since necessarily expert systems can be concerned only with some small aspect of the real world, which itself is considered decoupled from the rest of the world, and where all concepts that are frequently referenced can be primitives. The advantage is obvious: possibly with some effort, non-primitive nodes can now be *expressed in* (expanded to) primitives only; hence, if the (values of the) primitives are known, all nodes can be explicitly 'evaluated'.

---

[1] In Quillian's first paper, nets had just one type of link, with the implicit meaning: 'is used in the definition of'; later researchers started to distinguish several types of links or relations.

Another increase in usefulness originated when different *types* of links were introduced, each with its own interpretation[1]. A further development was the introduction of additional information into the nodes: nodes thus became *frames* [Brachman, 1983].

We will use Michie's example, earlier described in section 4.3, to show the translation of a problem into a graphical representation of a net. The problem was:

Given: A, B, C, D and E are primitives
F = A and B; G = C and D; H = E; J = B and G;
K = G and E; X = (F and H) or (J and K).

Wanted: X.

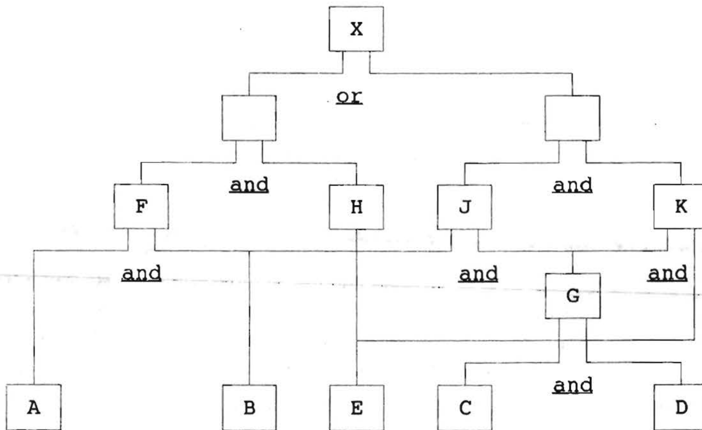


Figure 6.1. A graphical representation of an AND-OR net.

It is customary to draw a net as an inverted tree-like structure, with the root at the top. As drawn, the net is an AND-OR net, in which only two-input *and* and *or* operators occur; these operator symbols are sometimes replaced by special graphical symbols. It is also customary to split up complex expressions; in the example, this results in the two 'anonymous' nodes below X.

The links are *directed*. The link from F down to A indicates that, in order to evaluate F, the conclusion of A is needed. The symbol *and* below F indicates, that the conclusion of F is obtained by applying the *and* operator to the conclusions of the nodes below it.

The net is not a tree because nodes can be referred to more than once by higher level nodes; in figure 6.1, for instance, B is needed in the evaluation of both F and J.

---

[1] The interpretation of a link is a complex issue, however [Brachman, 1983].

6.3. Speed aspects

In SIMPLEXYS, run time searching for the relations between concepts is avoided completely because, besides storing the information embodied in the concepts, the links associated with them are stored as well (section 5.6.1). A prerequisite for this is, that the knowledge base (the rule set) is fixed, and that it is also known beforehand how to start the inferencing, i.e. which conclusions are to be derived. Then all rules and all links between rules are known and available for analysis, and they can be built into an internal representation of a net through a process called *compilation*, which assigns them known memory locations. Then, when a rule is evaluated, sub-rules (in 'backward chaining') or 'next' rules (in 'forward chaining') need not be searched for, but can be directly found.

Checking for errors in the knowledge base at run time decreases the inferencing efficiency. Although as much run time consistency checking is performed in SIMPLEXYS expert systems as efficiency allows, full run time checking is not compatible with efficiency. Moreover, run time checking, even if fairly exhaustive, will generally not discover all errors. Run time error checks are warranted only if errors of that type cannot be discovered at compile time.

6.4. Safety aspects

Before we explain how the knowledge in SIMPLEXYS expert systems can be verified, it is necessary to briefly review how knowledge is represented in such systems or, more accurately, how the evaluation of rules is performed. Rules can obtain a conclusion in either of two ways: by *evaluation* of the rule or by *assignment* of a conclusion as a result of the evaluation of another rule.

6.4.1. Rule evaluation

Two types of rules exist, primitive rules and composite rules. Evaluation of a primitive rule is a direct evaluation of its conclusion by some procedure, independently from other rules. A composite rule, also called an evaluation rule, needs the conclusions of other rules; it has an *evaluation expression* which expresses the way in which the rule acquires its conclusion if it must be evaluated.

Evaluation rules get their conclusion by evaluating their constituent rules; when those rules are composite rules themselves, these are also evaluated. This recursion process ends when primitive rules are reached.

6.4.2. Conclusion assignment

The SIMPLEXYS syntax allows multiple conclusions from a single evaluation. Whenever a rule is evaluated, THELSEs may give conclusions to other rules based upon the previous evaluation. This assignment is automatic and is considered part of the primary rule's evaluation. For example:

```
R1: THEN FA R2                    R3: ELSE TR R4
```

The first example states that R1 implies NOT R2. The second example, likewise, that NOT R3 implies R4. These assignments can, in turn, produce new assignments, e.g. R2: ELSE TR R5; i.e. assignments recursively propagate as well.

6.5. Systematic checking of a knowledge base

6.5.1. Checking for completeness

An important aspect of systematic checking of the knowledge base is a check for completeness: does the knowledge base contain all the rules necessary to produce the desired conclusions?

One approach to identify missing rules would be to assume that for all possible combinations of domain variables a rule should apply. However, there may be a great many of such combinations, most of these combinations would probably not be meaningful and therefore deep semantic knowledge of the meaning of the rules would be required before this task could become of practical significance. In general, the program that must check the knowledge base has no such deep knowledge of the problem domain and is thus not capable of checking the deep semantics of the rules. Therefore such a program cannot guarantee a completely correct knowledge base; it can only improve our confidence in a correct behavior.

SIMPLEXYS does not check for completeness other than for missing rules, i.e. rules which are referenced but not defined (see section 5.6.1)[1].

---

[1] Rules which are defined but not referenced are also detected by the Rule Compiler.

6.5.2. Checking for consistency

In a rule based knowledge base, errors at the rule level [Suwa et al, 1984] take the form of:

1. *Conflicts.* A conflict arises if, when two or more evaluation paths exist for one conclusion, those evaluations can give conflicting results. This situation can lead to inconsistent or even erroneous behavior of the expert system.

2. *Redundancies.* Redundancy arises when two or more rules always necessarily result in the same conclusion. Although redundancy normally does not cause erroneous behavior (this depends on the knowledge engineering language), it points out a possible implementation error. At least it indicates that the knowledge base can be simplified.

Similar errors can occur within a single rule:

1. A *contradiction* arises within a rule's expression if the rule's conclusion is always *false*, regardless of the situation in which it is applied. If the contradiction is not the result of an implementation error, such rules are superfluous and can be removed.

2. A *tautology* occurs within a rule's expression if the rule's conclusion is always *true*, regardless of the situation in which it is applied. If the tautology is not the result of an implementation error, such rules are superfluous and can be removed. A *partial* tautology occurs within a rule if its conclusion does not depend on one or more of its terms. If the partial tautology is not the result of an implementation error, such rules can be simplified.

Contradictions and tautologies are very similar types of redundancies. Because they are treated alike in all tests we will, in the remainder of this chapter, call either a *tautology.*

The following sections will discuss the above types of errors in SIMPLEXYS knowledge bases, but we first describe two valuable tools.

6.5.2.1. Semi-symbolic evaluation and Quine's method

In the intermediate stages of the process of error detection, two generally applicable discovery mechanisms play an important role, called *semi-symbolic evaluation* and *Quine's method*. This section discusses these mechanisms.

Assume that a rule has an evaluation expression 'R1 AND R2'. Also assume that it is known that R1 always has the conclusion TR. We can enter this knowledge into the expression by writing it as 'TR AND R2'. Such an expression, that includes at least one

constant value (TR or FA), is called a *semi-symbolic expression*[1]. Many of the correctness tests use a procedure that simplifies semi-symbolic expressions; the result of the simplification is either a fully symbolic expressions (in the example above: 'R2') or a constant value (TR or FA). An example of the latter is when the expression is again 'R1 AND R2', but now R1 is known to be FA; this results in 'FA AND R2', which can be simplified to 'FA'.

Examples of semi-symbolic expressions and their simplifications are:

```
E1: FA AND E3 OR (FA AND P2) OR P3        ->    E1: P3
E2: E1 AND E3 OR (P1 AND P2) OR TR        ->    E2: TR
```

The semi-symbolic evaluator uses eight very simple simplification rules. These are:

```
TR  AND ANY -> ANY                  TR  OR  ANY -> TR
ANY AND TR  -> ANY                  ANY OR  TR  -> TR
FA  AND ANY -> FA                   FA  OR  ANY -> ANY
ANY AND FA  -> FA                   ANY OR  FA  -> ANY
```

where ANY represents any expression. More sophisticated simplification rules like P1 OR NOT P1 -> TR can be shown to be superfluous.

Another procedure used in many of the tests is one that can (partially) simplify *symbolic expressions*; the result of the simplification is again either a fully symbolic expressions or a constant value (TR or FA). This procedure is based on Quine's truth-value analysis [Quine, 1958], to be referred to as *Quine's method*. This method implements the following idea:

- select a variable that occurs in the expression;
- replace that variable by the value TR, use the semi-symbolic evaluator to simplify the resulting expression, and note the (symbolic or constant) result;
- replace the same variable by the value FA, use the semi-symbolic evaluator again to simplify the resulting expression, and note the second (symbolic or constant) result;
- if both results are the same[2], that variable is redundant.

The following example demonstrates that expression E1 is independent of the term P1:

```
E1: P1 OR P2 OR NOT P3 OR NOT P1

P1 = TR -> E1: TR OR P2 OR NOT P3 OR FA     ->    E1: TR
P1 = FA -> E1: FA OR P2 OR NOT P3 OR TR     ->    E1: TR
```

---

[1] Note that the SIMPLEXYS syntax does not allow such expressions; they always represent an intermediate step in the checking process.
[2] Checking for logical equivalence of two expressions is described below.

This example does not only show that E1 is independent of P1, but also that expression E1 is a tautology. A necessary condition for a symbolic expression to be a tautology (have a constant value, either TR or FA) is that some term in that expression occurs both in the positive and negative form. A term appears in negative form when it is part of a NOT expression, e.g. NOT P1, or when that term belongs to a sub-expression that is preceded by a NOT operator, e.g. NOT (P1 OR P2), where both P1 and P2 are in negative form.

Generally, one application of Quine's method simplifies an expression by eliminating one redundant variable. Multiple applications can of course eliminate all redundant variables from the expression. A good substitution strategy is to choose the variable that has the greatest number of repetitions and also occurs both in the positive and negative form; this strategy tends to hasten the disappearance of variables, and thus to minimize the work. The expression is a tautology if all sub-expressions evaluate to the same constant value (TR or FA; using this method, the value PO will not arise).

Another use for Quine's method is to check whether an expression can obtain a certain value. For example, if

```
E1: P2 AND P3              and              E3: P4 AND NOT P2
```

are given, E1 will always have value FA and is thus a tautology. But in

```
E1: (P2 AND P3) OR P5      and              E3: P4 AND NOT P2
```

E1 can have both values FA and TR, the latter if P5 has value TR.

Checking whether two expressions are logically equivalent is done with Quine's method as well. First symbolically simplify both expressions to remove redundant variables. A necessary condition for the two expressions to be logically equivalent is now that both expressions contain exactly the same variables. Replacing each variable in turn by both TR and FA in both expressions and semi-symbolically simplifying the results builds two trees all of whose leafs will match if the expressions are logically equivalent.

6.5.2.2. Checking for conflicts

Checking the SIMPLEXYS knowledge base is done by several passes of the Rule Compiler; the first pass checks for syntactic errors, later passes for semantic errors. Because the logical independence of the set of primitive rules (e.g. TEST rules that tests externally supplied data) cannot be guaranteed (for an example, see section 6.5.2.3.1), the removal of redundancies cannot be complete, but otherwise a SIMPLEXYS knowledge base is well suited for consistency checking and especially for conflict checking.

The conclusion of a rule must be consistent with the conclusions of all related rules. Whenever, during an expert system's operation, a rule obtains a defined conclusion (TR, FA or PO), that conclusion is not allowed to change again during the same run. For example, a THELSE-construction trying to assign conclusion FA to a rule that already has conclusion TR, causes a run time error. Such an error is normally due to an inconsistency in the knowledge base and should be repaired by the knowledge engineer. We can often prevent such run time errors by trying to discover them at compile time. The Rule Compiler detects conflicts in the knowledge base by looking for possible erroneous interactions among rules.

We now take a closer look at the way in which the presence of conflicts in a knowledge base is checked. Full checking is not possible, because no deep knowledge of the domain is available, but many logical conflicts can be detected.

6.5.2.2.1. Detection of circular definitions

An evaluation rule usually gets its conclusion through backward chaining or evaluation. The evaluation is conditional, i.e. the evaluation stops as soon as the conclusion of the expression cannot change anymore, but in the worst case all operands must be evaluated. An evaluation can end only if its expression is fully reducible to primitive rules. A never ending evaluation appears only if the rule somehow references itself. In that case the net contains a loop[1]. A program checks whether each evaluation rule is a function of primitive rules only. This is done by building a *connectivity matrix*, where matrix entry [i, j] indicates whether rule i (a rule of any type) occurs in rule j's expression (this is meaningful only if rule j is an evaluation rule). Whether a rule uses another rule directly can be found in the rule's tokenized expression (see appendix 4); this information is entered into the matrix. Whether a rule uses another rule through any number of recursions can be decided by building the transitive closure, using the following algorithm:

```
procedure closure;
var
  i, j, k: 1 .. number_of_rules;
begin
  for i := 1 to number_of_rules do
    for j := 1 to number_of_rules do
      if connected [i, j] then
        for k := 1 to number_of_rules do
          if connected [j, k] then
            connected [i, k] := true
end {closure};
```

Figure 6.2 shows the relevant parts of the connectivity matrix before and after application of procedure closure, where the matrix is initially constructed from:

---

[1] In graph-theory terminology: a correct net has to form a directed acyclic graph.

```
R1: R2
R2: R3 AND R4
```

	R1	R2	R3	R4
R1		*		
R2			*	*

	R1	R2	R3	R4
R1		*	*	*
R2			*	*

Figure 6.2. The *uses* connectivity matrix before and after closure.

Loops in the knowledge base are detected by checking the diagonal entries of the matrix. This test is rather trivial, but is needed as a basis for further checking. An example of a loop:

```
E1: E2 OR E3
E2: P1 AND P2
E3: P3 OR E1
   -> E1: (P1 AND P2) OR (P3 OR E1)
```

Note that without this check it is not unthinkable that an expert system containing such a loop operates without error for a long time (e.g. because P3 is almost always TR) and then suddenly breaks down completely.

6.5.2.2.2. Detection of circular THELSEs

THELSEs can also form loops, but these loops are of a different nature. The evaluation of an expression is a form of backward chaining whereas THELSEs are a form of forward chaining. A problem can arise if a THELSE of a rule starts a chain or sequence of THELSEs that includes the rule that started the chain. This THELSE loop is illegal if a rule, that starts a chain of THELSEs is given a conflicting conclusion by that chain. In real-world knowledge bases, these chains can be hidden by their length. Testing a knowledge base by observing the system's behavior in a number of test cases does not always reveal these errors. 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 conclusion TR, also tries to give it the conclusion FA. To detect such conflicts, a THELSEs connectivity matrix is built where each matrix entry denotes the THELSE operation (e.g. THEN TR or ELSE FA), and another transitive closure procedure is executed, which forms a combination through appropriate matching. Two matchings are necessary in the previous example:

```
R1: THEN TR R2; R2: THEN FA R3
          |  |__|       |
          |_____________|

R1: THEN                FA R3
```

and subsequently

```
R1: THEN FA R3; R3: ELSE FA R1

R1: THEN                FA R1
```

The latter combination is the one that shows the conflict. Figure 6.3 shows the relevant parts of the THELSEs connectivity matrix before and after the closure. The error is discovered by noticing that after the closure a diagonal entry has a THEN FA.

	R1	R2	R3
R1		TT	
R2			TF
R3	EF		

	R1	R2	R3
R1	TF	TT	TF
R2	TF		TF
R3	EF		

Figure 6.3. The *THELSEs* connectivity matrix before and after closure.

If the example is modified into

```
R1: THEN TR R2;    R2: THEN FA R3;    R3: ELSE TR R1
```

there is still a loop, but this loop is harmless (and legal); a diagonal entry 'THEN TR' in the THELSEs connectivity matrix does not signify an error.

6.5.2.2.3. Detection of THELSEs to successors

A rule's evaluation expression contains references to other rules, that must be evaluated to give the rule its conclusion. It is not logical for the former rule to also assign a conclusion to those latter rules. If a rule *needs information* from rules lower in the net, it cannot also already *know* the conclusions of those rules. The connectivity matrices provide the information for these checks. An example:

```
E1: E2 AND E3            E1: THEN TR E2
```

After the evaluation of E1, during which E2 was evaluated, E2 is assigned a conclusion, which might conflict with the evaluated conclusion.

6.5.2.2.4. Detection of THELSEs to predecessors

A rule's evaluation expression contains references to another rule, lower in the net. It is then not logical for the rule lower in the net to assign a conclusion to the rule above it. If a rule *needs information* from rules below it in the net, the lower rule cannot also *know* the

conclusion of one above it. The connectivity matrices again provide the information for these checks. An example:

```
E1: E2 AND E3              E2: THEN TR E1
```

During the evaluation of E1, E1 is assigned a conclusion when E2 is evaluated, which might conflict with the conclusion that the evaluation will later return[1].

6.5.2.2.5. Detection of conflicting THELSEs

6.5.2.2.5.1. Checking THELSE operands

A THELSE construction consists of an operator, e.g. THEN TR, and two operands, the left-hand side and the right-hand side of the THELSE. The rule on the left-hand side is called the source rule and the rule(s) on the right-hand side the target rule(s). Whenever the source rule and a target rule expressions refer to common rules, that THELSE should be checked; this is a generalization of the checks of sections 6.5.2.2.3 and 6.5.2.2.4. A problem occurs if a combination of common variables exists so that both rules get conclusions that conflict with the THELSE. Consider the following situation:

```
E1: P1 AND P2
E2: NOT P1 AND P2 AND P3
E1: THEN TR E2
```

E1 is the source rule, E2 the target rule of the THELSE. Primitive rules P1 and P2 are the common variables, common to both E1 and E2. If both P1 and P2 are TR, an error can occur (and if P1 and P2 are independent, the error *will* occur once in a while at run time). When E2 is evaluated first and gets the conclusion FA, then evaluating E1 gives an error: E1 evaluates to TR and activates the THELSE that tries to give E2 a conflicting conclusion.

Checking the THELSE combination is done as follows. Source and target rule expressions, each fully decomposed into primitive rules, are combined into one expression. This expression is constructed in such a way that when it is subjected to Quine's method and a condition can be found that makes its value TR, an erroneous THELSE is found. The combinations are as follows:

```
R1: THEN TR R2     ->     R1 AND NOT R2
R1: THEN FA R2     ->     R1 AND R2
R1: ELSE TR R2     ->     NOT R1 AND NOT R2
R1: ELSE FA R2     ->     NOT R1 AND R2
```

[1] The Inference Engine also checks for this error at run time; it does not allow assigning to a rule while that rule is in the process of being evaluated.

The THELSE construction needs to be checked only if the expressions of source and target rule have common primitives. Using Quine's method, one of the common variables is selected and replaced by both TR and FA, giving two sub-expressions. The checking of a sub-expression ends if that sub-expression becomes FA after simplification or if it contains no more common primitives. All sub-expressions are checked this way and whenever one evaluates to TR, an erroneous THELSE construction is found. When the two operands have no primitive rules in common, nothing can be checked. If all primitive rules are independent, this check is complete. An example of the method:

```
Given  E1: (P1 AND P2) OR ( P3 OR P4 OR P5 OR P6)
       E2: (P5 AND P6) OR P7
       E1: THEN FA E2
```

Common primitives: P5, P6

Combined expression: E1 AND E2

```
If P5 = TR and P6 = TR then
   E1 = TR and E2 = TR so
     (E1 AND E2) = TR -> CONFLICTING THELSE
```

6.5.2.2.5.2. Checking conflicting THELSE chains

Assignments can propagate new assignments. One of the consequences can be that a source rule can give a conclusion to another rule in more than one way. This situation occurs if two or more chains of THELSEs exist from one rule to another. In order to check if those chains of THELSEs have the same effect, the THELSEs connectivity matrix is inspected. Consider the following THELSEs:

```
R1: THEN TR R2           R2: THEN FA R4
R1: THEN FA R3           R3: ELSE TR R4
```

This set of THELSEs has two conflicting chains of THELSEs:

```
R1: THEN TR R2; R2: THEN FA R4 -> R1: THEN FA R4
R1: THEN FA R3; R3: ELSE TR R4 -> R1: THEN TR R4
```

All chains are automatically obtained when the transitive closure of the THELSEs connectivity matrix is built. Every new chain is compared with the chains found before, and conflicts are signalled.

6.5.2.2.5.3. Checking THELSE common targets

If two different rules both have a THELSE to a common rule, and if both THELSEs give the common target rule a different conclusion, then this situation must be checked. An error

occurs when both rules can get conclusions such that their THELSEs will conflict. For example:

```
R1: THEN TR R3            R2: THEN FA R3
```

When both R1 and R2 are TR, the two THELSEs conflict. The test is, again, to combine the two expressions into one, and to use Quine's method to check if a combination of conclusions of the common primitives exists so that both THELSEs will fire. In the example this means a combination of conclusions making both R1 and R2 TR. When both rules have no primitive rules in common, nothing can be checked. The two expressions are combined 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
E1: ELSE TR E3; E2: ELSE FA E3  ->  NOT R1 AND NOT R2
```

6.5.2.3. Checking for redundancy

6.5.2.3.1. Detection of tautologies

A tautology is an expression whose value after evaluation is always the same, regardless of the values of the operands. Finding a tautology does not automatically mean that a true conflict is detected. A tautology will not, for example, cause a run time error. Nevertheless we check for tautologies, because a tautology is usually caused by a knowledge base implementation error. It is peculiar to define a rule in such a way that the conclusion of that rule is independent of the expression's operands. In most cases a tautology is caused by a wrong combination of primitives, probably by a misinterpretation of the expert's knowledge. If this tautology is caused by an implementation error, the expression must be repaired. Otherwise the rule can be replaced by a constant conclusion, thereby simplifying the knowledge base: the rule can be removed and all expression rules that refer to this constant rule can be simplified or, in turn, eliminated as well.

Looking for tautologies means checking all evaluation rules after they are symbolically evaluated into functions of primitive rules only. Tautology checking can be done in several ways; the simplest method is by building and checking the expression's truth table, but because the number of truth table entries is exponential in the number of variables, this method is not practical; it is normally much faster to use Quine's method. Generally, the substitution process in Quine's method has to be repeated only a few times, each time substituting another variable. The process stops, concluding that the expression is not 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 cannot be simplified further; such an expression cannot represent a tautology. For more details, see Lutgens [1989].

It is obvious that an expression that is a tautology because of the semantics of the *primitives* is not found by the tautology checker. An example of two dependent primitive (Boolean TEST) rules:

```
P1: BTEST age > 30      and       P2: BTEST age <= 30
```

To allow more comprehensive checking, the rule base builder should remove rule P2 and replace all references to it by NOT P1.

A history operator and a history expression connected to a rule are also considered a primitive. Here a similar problem exists: the checker has no access to the Pascal code. For example, if the history expression 'R1 > (3)' occurs more than once in the knowledge base, the checker will consider all its occurrences to be different entities. Thus an expression like

```
(R1 > (3)) OR NOT (R1 > (3))
```

will not be discovered to be tautological. To allow more comprehensive checking, the rule base builder should create a new evaluation rule which contains the history expression only, and replace all occurrences of the history expression by a reference to the new rule.

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, as much as possible, primitives are indeed independent, so that the relations between chunks of knowledge are visible to the checker. This means that, for instance, no primitive rule should exist that can be expressed in other primitives.

6.5.2.3.2. Detection of partial tautologies

Partial tautologies can also occur. A partial tautology occurs if not the evaluation expression as a whole is a tautology, but only part of it. For example:

```
P1 AND (P2 OR NOT P2)  ->  P1 AND TR  ->  P1
```

The expression can be simplified by the semi-symbolic evaluator. One way to find partial tautologies is by Quine's Method: choose one variable and make two expressions, one with TR and one with FA instead of the variable. The expression is independent of that variable if both resulting expressions can be simplified to equivalent expressions.

Partial tautologies can occur even if a term does not occur in both positive and negative form, as in

```
P1 OR (P1 AND P2)  -> P1
```

This makes finding partial tautologies a more time-consuming activity than finding tautologies.

6.5.2.3.3. Completion of expressions

Finding tautologies and partial tautologies is important. In SIMPLEXYS expressions, problems can arise with the operators *and* and *or* in expressions that are tautologies in two-valued logic, such as

```
p and not p          and          p or not p
```

In standard two-valued logics, the results are always *false* and *true*, respectively, regardless of the value of p. We would expect the same in three-valued logic. This is not true. If p has value PO, the run-time *evaluation* of the expressions yields *PO and not PO = PO or not PO* = PO, which is counter-intuitive and may be considered a defect of the SIMPLEXYS logic. In contrast, the evaluations are intuitively correct in the expressions

```
p and not q          and          p or not q
```

The problem is introduced because in the first two expressions the terms *p* and *not p* are *correlated* (they are each other's logical inverse). In other words: two-valued logic yields the intuitively obvious simplification rules

```
p and not p -> FA
p or  not p -> TR
```

that are often used in *symbolic* evaluations of (i.e. substitutions in) logical expressions. Intuitively, these rules should also apply in SIMPLEXYS logic. However, the *semi-symbolic evaluation* that we treated in section 6.5.2.1 yields the correct results, whereas the Inference Engine's *value evaluation* does not. This discrepancy is unsatisfying, and must be repaired.

First we must recognize all tautologies in a knowledge base, because they usually are errors. If either *p and not p* or *p or not p* occur in a logical expression (possibly after expansion of that expression into primitives), this is probably an implementation error because the *semantics* of the expression are unclear and resemble nonsensical sayings like 'it rains but it does not rain' or superfluous sayings like 'it rains or it does not rain'. If such an expression occurs in a knowledge base, it must therefore be found and signaled to the user as a probable error.

But the problem is more pervasive, because many expressions can result in a tautological form in some special cases. Let us restate the problem in general: evaluation by the Inference Engine should give the same results as semi-symbolic evaluation would. This is achieved as follows. If an expression references a term X both in positive and negative form

(this can be verified by expanding that expression into its primitives), we can always rewrite that expression as:

```
(X and E1) or (not X and E2) or E3
```

with truth table

E1	E2	E3	X=TR	X=PO	X=FA	
TR	TR	TR	TR	TR	TR	
TR	TR	FA	TR	PO	TR	<- error
TR	FA	TR	TR	TR	TR	
TR	FA	FA	TR	PO	FA	
FA	TR	TR	TR	TR	TR	
FA	TR	FA	FA	PO	TR	
FA	FA	TR	TR	TR	TR	
FA	FA	FA	FA	FA	FA	

The Inference Engine would generate an erroneous outcome when E1 = E2 = TR and E3 = FA; in that case the expression reduces to the tautology *X or not X*. Since both X = TR and X = FA yield a result TR, X = PO should of course give result TR as well. This observation immediately suggests a solution, however: expand the expression that the Inference Engine must evaluate into:

```
(X and E1) or (not X and E2) or E3 or (E1 and E2)
```

Logically, the expansion is superfluous, but with it the Inference Engine's evaluation proceeds correctly for all values of X. If the expression contains more terms in both positive and negative form, extra expansions for these terms must be introduced as well. With this procedure, a correct result is obtained in all cases.

Modal logic [e.g. Hughes and Creswell, 1968; Chellaas, 1980] offers some insight into this problem. Besides the truth values *true* and *false*, it has the 'modifiers' *possibly* and *necessarily*. An expression is *necessarily true* if it is true regardless of the conditions; we have called such an expression a *tautology*, and an expression that is *necessarily false* a *contradiction*. Modal logic learns us, that in order to establish whether an expression is *possibly true* or *necessarily true* requires a *global* evaluation, which may need to take into account many other items in the knowledge base; we have indeed seen, that in either case we need to expand the expression into its primitives. But whereas whether an expression is a tautology can be established at compile time (because it is true regardless of the actually occurring conditions), the establishment of whether an expression is *possibly true* cannot be done at compile time (because this *does* depend on the actually occurring conditions).

In practice, the number of necessary expansions will often be small. Many expressions will not contain both positive and negative references to any rule. And if rule X is of a type that

cannot have value PO (BTEST or STATE), no expansion need take place. If an expansion is necessary, the extra evaluation effort will be small, since no extra rules need to be evaluated when the Inference Engine encounters the expansion; if at that point the value of the expression is either TR or FA, evaluation of the expansion need not take place at all. More shortcuts are possible. Instead of straightforwardly evaluating the expression's expansion

```
... or (E1 and E2)
```

we can perform the evaluation

```
if R = PO then
  if eval (E1) = TR then
    if eval (E2) = TR then R := TR;
```

where R is the result of the original evaluation and eval is a function that evaluates an expression. Since E1 and E2 have already been evaluated before, the necessary rule values can simply be looked up; not a single additional rule needs to be evaluated.

At this point, we cannot properly evaluate this 'expansion mechanism' since we have not enough insight into how often such expansions must be added to expressions in an average rule base, nor how much extra effort is necessary to evaluate an average expansion[1].

6.5.2.3.4. Elimination of FACT rules

A related but much less complicated situation will be encountered frequently, due to the probable occurrence in the knowledge base of primitive rules of type FACT. FACT rules have a constant conclusion during the expert system's analysis phase, because the conclusions of these rules are acquired before the expert system starts up. From that moment on, the knowledge base contains rules with constant conclusions. It is possible to eliminate all FACT rules and to semi-symbolically simplify the evaluation expressions containing these rules, thus simplifying the knowledge base.

6.5.2.3.5. Detection of equivalent rules

Redundancy means that a rule is superfluous because it is equivalent to another rule. As with tautologies, the SIMPLEXYS Inference Engine does not have any practical problems with redundancies; they do not lead to run time errors. Nevertheless, we check for redundant rules because, again, redundancy usually indicates an implementation error. If not, redundant rules can be removed from the knowledge base without affecting the operation of the expert system; the effect is a more compact knowledge base and thus a more efficient evaluation.

---

[1] This is because the only major size rule bases we have currently experience with do not employ the truth value PO and therefore do not require such expansions.

6.6. Limitations

A major limitation to checking is the fact, that the checker has no access to the Pascal code. It would be a major effort to incorporate a fully functional Pascal compiler into the Rule Compiler. However, doing so would allow several checks that so far are impossible; some problems have been mentioned above. Another series of errors can occur because SIMPLEXYS allows *any* Pascal code in TEST rules and DO sections. It is not hard to imagine a situation in which several TEST rules need the value of a Pascal variable, while other Pascal statements *change* the value of that variable. This makes the order of execution of the rules very important, and the result could be very unpredictable behavior.

In other respects, too, checking is not full proof. A good design methodology, an awareness of the limitations of his toolbox and a critical attitude with respect to the implemented knowledge are necessary assets for a knowledge engineer.

6.7. Conclusions

SIMPLEXYS allows better consistency checks than many other expert systems languages, most of all because the relations between rules are explicitly expressed in boolean-like expressions and by THELSE constructions. The consistency checks described here must be considered a first attempt to develop methods to ensure the consistency of SIMPLEXYS knowledge bases as much as possible. The currently implemented checks have proven to be very useful.

7. Checking the protocol

This chapter discusses the theory on which the SIMPLEXYS Protocol Checker is based. It describes protocols, their translation into Petri nets, what protocols denote, how they can be analyzed, and how they are used in SIMPLEXYS. Formal definitions of Petri nets can be found in the literature [Petri, 1986; Reisig, 1985]. Many different Petri net classes have been invented, all with common basics, but each with its own special features; the terminology that is used depends on the class of the considered net. SIMPLEXYS protocols correspond best with the Petri net class of what are called Live Safe Place/Transition nets [Lammers, 1990a], but we will use a terminology that reflects the use of these nets in SIMPLEXYS (*state* instead of *place*, *trigger* instead of *transition*).

Petri nets, first described in Petri's dissertation [Petri, 1962], are abstract data structures that specify or model the dynamic (time sequenced) behavior of systems or processes, such as message passing systems or computer networks. Since this first publication, Petri nets have become an important topic for research; the terminology has been formalized and many additional theoretical results have been derived. Petri nets are ideally suited to describe systems in which concurrent events can occur, i.e. systems in which some parts operate in parallel with other parts.

7.1. From ON statements to protocol

We recall that the syntax of an ON statement is:

```
ON trigger FROM list1 TO list2
```

where list1 is a list of STATE rules, to be called the FROM-list or pre-state list, and list2 is also a list of STATE rules, to be called the TO-list or post-state list.

The ON statements of a SIMPLEXYS knowledge base collectively define a protocol or Petri net (for the moment we will assume that protocols and Petri nets are identical). A Petri net consists of four sets of elements:

1. STATE rules or simply *states*, drawn as circles.
2. Trigger rules or simply *triggers*, drawn as boxes or bars.
3. Directed *arcs* that describe the relationship between the states and the triggers.
4. *Tokens*; tokens can be placed in states and removed again; the configuration of the placed tokens is called the 'marking' or 'context'. An operation of moving one or more tokens is called a 'context switch'.

Given the set of ON statements of a SIMPLEXYS knowledge base, the graphical representation of the protocol is constructed as follows. Draw a circle for every STATE rule

and for every '*' in the set of ON statements (i.e. every '*' is to be considered an additional, unique, anonymous STATE rule). Draw a box or bar for every trigger rule. For every ON statement, draw arcs *from* all states in the FROM (pre-state) list to that ON statement's trigger, and also arcs from that trigger *to* all states in the TO (post-state) list. Finally, put a token in all states that correspond with a STATE rule with an initial value true[1].

Figure 7.1 gives the graphical representation of a set of ON statements as a protocol[2].

ON statements:

```
ON t1 FROM s1 TO s2
ON t2 FROM s3 TO s6
ON t6 FROM s2 s6 TO s7
ON t3 FROM s3 TO s4
ON t4 FROM s4 TO s5
ON t5 FROM s5 TO s6
ON t1 FROM s7 TO *
```

s1 and s3 are STATE rules with initial value *true*

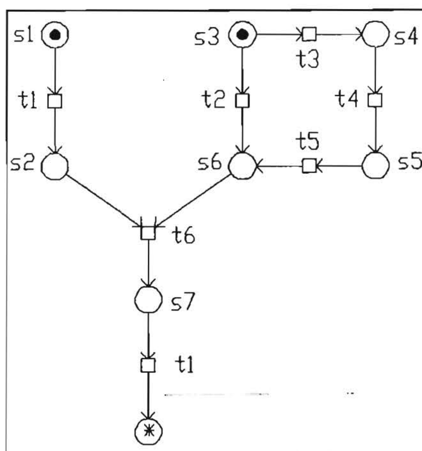


Figure 7.1. Graphical representation of ON statements.

Active states are marked with tokens. The set of states that is simultaneously active will be called the *context*. The *initial context* is the set of states that is active when the system starts up. In figure 7.1 the initial context consists of states s1 and s3.

A trigger is *enabled* if all states that have an arc to it are marked with a token. In figure 7.1, triggers t1, t2 and t3 are enabled, by the fact that s1 and s3 are marked. An enabled trigger can *fire*. When a trigger fires, one or more tokens migrate: all *pre-states* loose their tokens while all *post-states* obtain a token. Such a migration is called a change of state or *context switch*. Thus, in figure 7.1, when t1 fires, s1 looses its token while s2 gets one.

Each ON statement specifies a *trigger* and its pre- and post-states. The trigger's pre-states are those in the FROM list, and its post-states are those in the TO list. The trigger is

---

[1] In this chapter we write *true* and *false* rather than TR and FA; the value PO plays no role in protocols and Petri nets.

[2] This is a very small example. The graphical representation of most protocols would be far too large to display. The tracer/debugger tool (section 5.5.7) can display small selected parts of the protocol while debugging a knowledge base.

enabled if all its FROM list STATE rules have value *true*. If the trigger is enabled, the Inference Engine evaluates the rule connected to the trigger. If the rule evaluates to *true*, the trigger *fires* and we have a context switch: all STATE rules in the FROM list become *false* and all STATE rules in the TO list become *true*.

States that are in the initial context are called *initial states*. There are also *final states*, anonymous states represented by a '*'; this is concomitant with the notion that a protocol normally has a beginning and an end. The *final context* is defined as the context in which only final states have a token. When the Inference Engine detects the final context, it halts: the protocol has reached its end.

A context is called *reachable* if there is a sequence of ON statements, called a *firing sequence*, that when executed results in that context. A *reachability list* can be constructed that contains all contexts that are reachable from the initial context. The first entry of the list[1] is the initial context. In the initial context, usually several triggers are enabled. Firing one of the enabled triggers results in a successor context. Symbolically firing each of the enabled triggers in turn results in all possible successor contexts; these are added to the list if they are not already in it. These contexts can again be expanded to their successors, and these are also added to the list if they are not already in it. This way a list with all possible contexts can be constructed. In a valid protocol, the final context must of course be constructed as well. Figure 7.1's reachability list will have the following 10 entries:

```
s1 s3     initial context
s2 s3     t1 fired from initial context
s1 s6     t2 fired from initial context
s1 s4     t3 fired from initial context
s2 s6     t2 fired from s2 s3
s2 s4     t3 fired from s2 s3
s1 s5     t4 fired from s1 s4
s2 s5     t4 fired from s2 s4
s7        t6 fired from s2 s6
*         t1 fired from s7
```

One or more *goals* are usually connected to each state; these goal rules are evaluated when that state is *true*, i.e. belongs to the context. The relationship between the protocol and the remainder of the rule base is that the protocol defines the goals that must be evaluated, and that the remainder of the rule base delivers the values of the triggers which the protocol needs to change the context. States, triggers and goals play different roles in SIMPLEXYS: states represent (context) memory, triggers (context switch) actions, and goals conclusions to be derived in that context.

---

[1] We actually construct a *tree* rather than a *list*, in order to be able to present more helpful error messages.

7.2. Petri net basics

SIMPLEXYS protocols are not proper Live Safe Place/Transition nets[1], and Petri net theory cannot be applied to SIMPLEXYS protocols: proper Live Safe Place/Transition Petri nets have an 'endless' protocol. A SIMPLEXYS protocol can be converted into a net of the desired Petri net class, however, by extending it with a single extra trigger from all the final anonymous '*' states to all the initial states; this extra trigger 'closes the loop'. In a monitoring context this extra trigger would mean that the expert system is 'ready for the next patient'. We will not differentiate between such an 'extended SIMPLEXYS net' and its Petri net interpretation.

The relationship between Live Safe Place/Transition Petri nets and SIMPLEXYS protocols is not a strict one; SIMPLEXYS protocols are much more lenient (see section 7.4). Petri net theory is used as a *tool* to discover *possible errors*: erroneous or questionable steps in the protocol. A SIMPLEXYS protocol need not necessarily be live or safe in the Petri net theory sense, but if it does not have both these properties, it is constructed in an intuitively unobvious way, which might indicate an implementation error (but the protocol might also be correct but tricky, and therefore difficult to maintain).

An example of a regular Petri net is given in figure 7.2. In the SIMPLEXYS protocol, states s3 and s7 were anonymous final states, but for convenience these have been given names. Trigger t7 'extends' the SIMPLEXYS protocol into a Petri net.

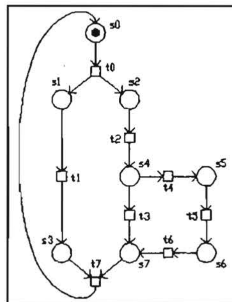


Figure 7.2. A regular Petri net.

The initial context consists of state s0 only. When t0 fires, s0 looses its token while s1 and s2 get one. This results in the context shown in figure 7.3.

---

[1] The definitions of *live* and *safe* are given at the end of this section, after the basic notions have been introduced that are used in these definitions.

In figure 7.3, a new situation exists; both t1 and t2 are now enabled because their pre-states s1 and s2 are marked. Triggers t1 and t2 can fire independently: if one fires, the other is still enabled. There is *concurrency*: the left branch of the net can operate in parallel with and independently of the right branch.

Figure 7.4 shows the result after both t1 and t2 have fired. Both t3 and t4 are enabled, but if t3 fires, t4 is no longer enabled and if t4 fires, t3 is no longer enabled. This is called a *choice*: the single token can move either along t3 or along t4, but not both ways. If two triggers that are in a choice fire at the same time, a situation that Petri net theory calls a *conflict* arises: it is not specified which way the token should go. Conflict checks are described in sections 7.3.2, 7.3.4 and 7.4.

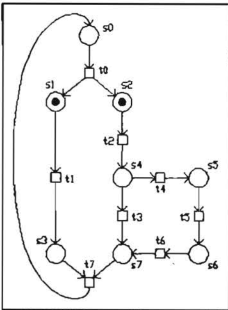


Figure 7.3. The context has switched FROM s0 TO s1, s2.

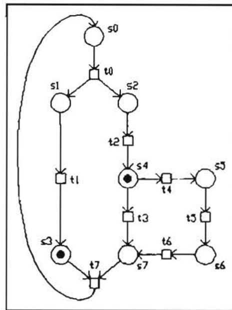


Figure 7.4. The context has switched FROM s1, s2 TO s3, s4.

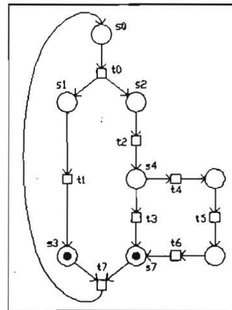


Figure 7.5. The context has switched TO s3, s7.

When trigger t4 fires, t5 is enabled; then, when t5 fires, t6 is enabled; the subsequent firing of t6 then results in the net shown in figure 7.5. This context is identical with the one that would occur if in figure 7.4 trigger t3 had fired.

In figure 7.5, t7 is enabled. When t7 fires, the net returns to its initial context of figure 7.2. It is a property of a correct Live Safe net, that such a return to the initial context is possible.

In Petri net theory, the *state capacity* of a state is the maximum number of tokens which that state can carry. Most often the capacity of every state is taken to be equal to one. This means that no state is allowed to have more than one token, and that either triggers can only fire if there is enough capacity to store the tokens transported on firing, or that a token gets lost when it merges with the one already present. In a message passing system, the term 'state capacity' makes sense; it is the number of messages a buffering node can contain. In SIMPLEXYS, the term looses its meaning; a state is either active or not.

A net is called *N-safe* if, due to its topology, every state can have at most N tokens. For short, 1-safe will be called *safe*. Firing a trigger that puts a token on a state that already has one is thus a violation against safeness. In an unsafe SIMPLEXYS protocol, a true STATE rule can be made true again. This may well be a protocol design error.

A net is called *live* if every trigger can be enabled from every reachable context by some firing sequence, i.e. there are no triggers that can never fire. In a non-live SIMPLEXYS protocol some trigger rule can never fire. This is a protocol design error: the trigger rule is superfluous.

The net has a *deadlock* if it has a reachable context in which none of the triggers is enabled: that context has no successor context, the tokens are stuck. This is a violation against liveness. In a protocol with a deadlock, no progress will be possible after the deadlock occurs; the protocol cannot end. This is an obvious error.

7.3. Systematic checking of the protocol

Computer tools are available [Feldbrugge and Jensen, 1986; Jensen, 1986] to check a net for having the properties of Live Safe Place/Transition nets, but adaptation of these tools to the SIMPLEXYS programming environment would be inadvisable: if the net is incorrect, error messages must be provided that are understandable for knowledge engineers who are not Petri net experts.

The goal of the analysis is to check whether the ON statements in a SIMPLEXYS rule base represent, when extended, a Live Safe net. If not, this probably indicates an error in the implementation of the logic or the ordering of the protocol's elementary operations. The analysis is divided into sections that check whether the net has the desired *properties* of *live* and *safe*. Live and safe are, however, complex properties (see section 7.3.4), that in turn depend on other properties which are easier to check. Therefore the checks are sequenced in such a way that easier checks precede more difficult ones. An additional advantage of this ordering is that more meaningful error or warning messages can be given.

7.3.1. Detection of syntax errors by the Rule Compiler

The Rule Compiler has already performed some elementary syntax checking on the rule base, including the ON statements which are part of it. According to the Rule Compiler, a set of ON statements is syntactically correct if

a. Every state has non-empty FROM and TO lists. A TO list may contain the special symbol '*', which stands for an anonymous *final state*.
b. The rules in the FROM and TO lists are STATE rules.
c. At least one STATE rule is initially *true*.

When the Rule Compiler finds the knowledge base syntactically correct, the protocol analyzer takes over. Semantic checking of the protocol is performed in three stages. The first stage performs some additional syntax checks (section 7.3.2), the next two stages check for correct topology (section 7.3.3) and dynamics (section 7.3.4) respectively.

7.3.2. Detection of other syntax errors

Sets of triggers and sets of states are important internal data structures of the protocol checker. Pre- and post-sets of a state are sets of triggers, pre- and the post-sets of a trigger are sets of states. A set can contain from zero up to all states or triggers that the net contains.

The ON statements are first transformed into the internal format by storing the pre-set and the post-set of every state and of every trigger. This allows a fast analysis and the use of algorithms that are similar to the Petri net tools. During the transformation a first check is performed: the (extended) net must not have empty pre- and post-sets; that can be shown to be a necessary condition for a Live Safe net.

The syntax check reports the following errors:

No final state. If there is no ON statement with a '*' in its TO list, the expert system can never stop.

State cannot become active. Every state, except initial states[1], must have at least one trigger leading to it. This means that every state must be in at least one TO list. If not, that state is *unreachable*; it can never become *true*.

State cannot become inactive. If a state has no trigger leading from it, once *true* it can never become *false*.

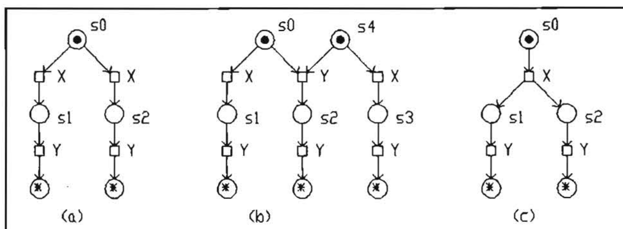


Figure 7.6.
a. Conflict at s0 with trigger X.
b. No conflict.
c. Both s1 and s2 get a token when X fires.

---

[1] Initial states are recognized by the fact that their initial value is *true*.

Conflict between triggers. More than one trigger may be enabled in a context. In figure 7.1 triggers t1, t2, and t3 are concurrently enabled. Triggers t1 and t2 can fire independently: if one fires, the other is still enabled. But t2 and t3 are dependent: firing one disables the other way. There is a *choice*, either to fire t2 or to fire t3, but not both. A problem exists when triggers t2 and t3 become *true* at the same time. It then depends on the textual order of the ON statements in the knowledge base which trigger will fire; this is undesirable because it may lead to maintenance problems.

Choices are unavoidable if the system is non-deterministic. Generally we will not know whether t2 and t3 can become *true* at the same time; if that happens, the choice becomes a *conflict*. At run time, the Inference Engine would handle the conflict by firing only one of the triggers. A conflict is certain, however, if two triggers are connected to the same rule and have identical FROM lists (figure 7.6a). One path, depending on the textual order of the ON statements, can never be taken. This obviously warrants an error message.

Figure 7.6b gives a correct net. In this net triggers X and Y occur in more than one ON statement, but no conflict occurs. In the net in figure 7.6a, there are identical FROM lists and identical triggers. If, in figure 7.6a, it was the intention to give both s1 and s2 a token after X becomes *true*, figure 7.6c is the correct representation.

7.3.3. Detection of topological errors

In the second stage topological checking is performed: the net must be *pure*, *simple* and *strongly connected*.

The net is called *pure* if there is no arc from a trigger to a state if there is an arc from that state to that trigger, i.e. there is no ON statement in which FROM and TO lists have states in common, i.e. there is no loop from a state to that same state (figure 7.7 shows two non-pure nets). A non-pure net is likely to be neither live nor safe, but this is not certain. A non-pure net is quite intricate and therefore warnings are generated for non-pure ON statements.

The net is called *simple* if there are no triggers with identical pre- and post-states and no states with identical pre- and post-triggers. This must hold for all states and triggers. Simple implies that no two net elements are functionally identical. Nothing is wrong with a non-simple net element, but it is likely that a mistake has been made. Moreover, the net can be simplified by combining identical net elements. For non-simple nets warnings are therefore generated.

Finally strong connectedness is checked. The net is called *strongly connected* if every net element can be reached from every other net element by walking along arcs; every state can be reached and left again. A net that is not strongly connected will not pass the checks of liveness and safeness (strong connectedness is a necessary condition for liveness and

safeness). By checking connectedness we report some errors in an earlier stage with more precise messages.

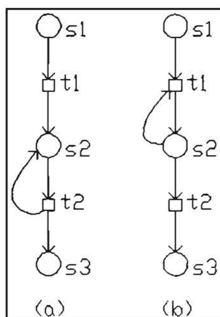


Figure 7.7.
a. Self loop at t2.
b. Self loop at t1.

The topological check reports the following errors:

Self loop. A self loop exists if ON statements have common states in their FROM and TO lists. In figure 7.7a state s2, once *true*, can never become *false*. Figure 7.7b gives the dual situation: s2 must be *true* before t1 can fire, but before s2 can be *true* t1 must fire. ON statements that have a self loop are likely to result in a deadlock or in a non-safe state. But this is not certain: correct nets with a self loop do exist. Self loops are therefore reported as a warning.

Identical ON statements. Two ON statements are identical if they have the same FROM and TO lists. This is not an error; it is reported as a warning. Identical ON statements can be merged by or-ing their triggers.

Identical states. Two states are identical if they have the same triggers leading to and going from them. This, too, is a warning. Two identical states can be merged by discarding one. Goals are taken together and connected to the remaining state.

State not connected with an initial state. There must be a path from one of the initial states to every state. A net part that is not connected by arcs to an initial state can never get tokens.

State not connected with a final state. Every state must have a path to a final state. A net part that is not connected to a final state can never loose its tokens.

7.3.4. Detection of dynamics errors

The final stage checks for dynamics errors. The reachability list is constructed and used to prove liveness and safeness. In this stage more complex conflicts are also checked. Three errors can occur: a deadlock, a non-safe state and a conflict.

The net must be *safe*. It is safe if firing a trigger never results in a state getting more than one token; in a safe net firing an ON statement will not cause a state to become *true* if it was *true*.

The net must be *live*. It is live if from every reachable context a firing sequence exists that results in a context that enables an arbitrarily chosen trigger.

When all reachable state combinations have been constructed, two more errors can be detected: triggers that did not fire, and non-reachability of the final context.

The dynamics check reports the following errors:

Deadlock. A deadlocked context does not enable any trigger. There is a firing sequence that results in a context where no further change of state is possible. Figure 7.8a gives a net that comes to a deadlock: in the initial context t1 or t2 fires, giving s1 or s2 a token. This token flows to s3 or s4. Then we have a deadlock: to fire t5, both s3 and s4 must have a token, whereas only either s3 *or* s4 can have one. The intended net was probably the one shown in figure 7.8b.

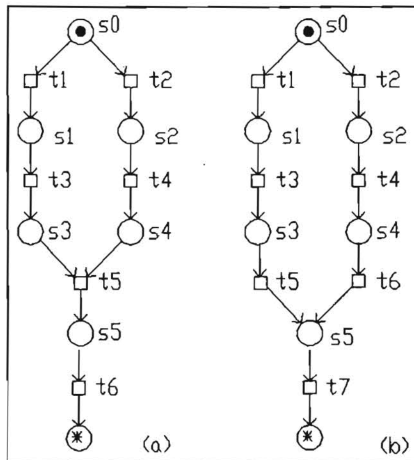


Figure 7.8.
a. A deadlock at t5.
b. A correct net.

Non-safe state. A state becomes *true* due to firing of a trigger while that state was already *true* before firing (and is not made *false* by another trigger firing simultaneously). This happens in figure 7.9a: when t1 fires, both s1 and s2 get a token. These tokens flow to s3 and s4. Then firing t5 gives s5 a token and firing t6 gives s5 another one. Therefore s5 is not safe: it becomes *true* while it was already *true*. Figure 7.9b is probably the intended net.

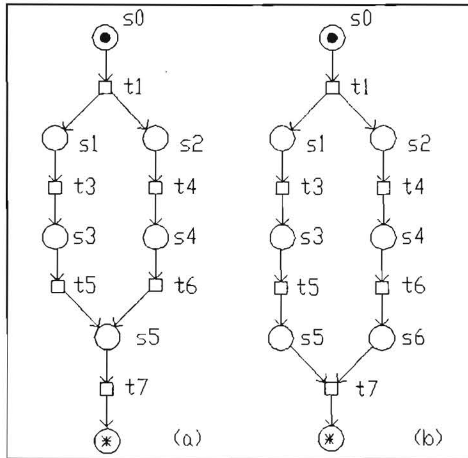


Figure 7.9.
a. State s5 can get two tokens.
b. A correct net.

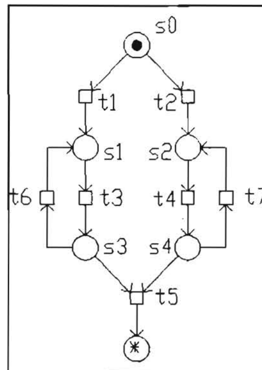


Figure 7.10. This net cannot stop.

The system cannot stop. There is no firing sequence that leads to only final states being *true*. Because the Inference Engine does not stop as long as at least one state is active, the system will never stop. It is possible to have a net that does not come to a deadlock while

the final context is not reachable; in figure 7.10, t5 can never fire. This net can be corrected in the same way as the net in figure 7.8.

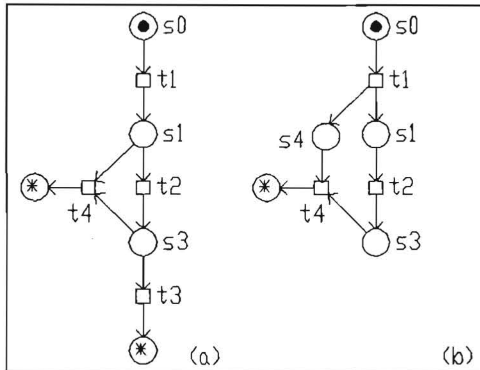


Figure 7.11.
a. Trigger t4 cannot fire.
b. All trigger can fire at least once.

Trigger cannot fire. Figure 7.11a gives a net that does not come to a deadlock, while the final context is reachable as well. However, there are triggers that cannot fire at least once.

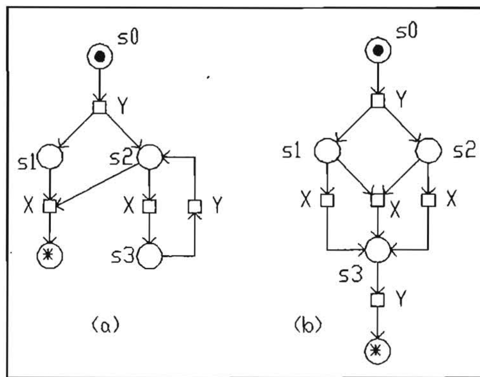


Figure 7.12.
a. Conflict between two triggers X.
b. Conflict between three triggers X.

Conflict. In the syntax checking stage some conflicts have already been checked: ON statements that have the same trigger rules and the same FROM lists. Conflicts can also arise when the FROM lists have a non empty intersection (figure 7.12a). In context s1 s2 both triggers, connected to the same trigger rule X, are enabled and must necessarily fire

simultaneously. But if the first has fired, the second cannot fire because s2 was made *false* by the first.

Whether or not two triggers lead to a conflict depends upon their trigger rules being *true* at the same time. We can generally not fully check for this during analysis. A conflict is easy to detect when two ON statements satisfy the following three conditions:

a. The two ON statements specify the same trigger rule;
b. The intersection of their FROM lists is not empty;
c. The two triggers are both enabled.

In the net of figure 7.12b there are three conflicting ON statements labeled with X.

Thus far, we have only reported on conflict checking where ON statements specify the same trigger rule. A conflict check can be more thorough, however. Recall that a conflict can only occur if triggers are *true* at the same time. To prevent this, we can replace the triggers by *trigger expressions* that mutually exclude each other. In figure 7.5, triggers t2 and t3 can lead to a conflict. If we replace t2 by t2 *and not* t3, and t3 by t3 *and not* t2, no conflict can possibly occur. However, in figure 7.6a such a substitution would lead to trigger expressions with value X *and not* X, i.e. *false*: the triggers would never fire; this is obviously incorrect.
This observation leads to the following check: use Quine's method (see section 6.5.2.1) to symbolically simplify the formed trigger expressions. Whenever this reduction leads to the constant value *false*, there is a conflict; if not, a conflict will occur whenever the simplified expression can obtain the value *false*. The knowledge engineer can check whether such conditions can actually occur and if so, whether they will lead to undesired behavior. The latter may depend on the order of the ON statements in the knowledge base.

7.4. Correctness checks at run time

When no errors are detected, the protocol has all the properties of and thus is a correct Live Safe Petri net. In many respects, the SIMPLEXYS Inference Engine is more tolerant than the protocol checker, but this does not guarantee that at run time no errors can occur. Whether or not the expert system can reach the final context depends upon the triggers that at run time become *true* in each context. The *structure* of a correct net guarantees that it will always be possible to continue and reach the end context after some firing sequence. However, whether this firing order is possible at run time depends upon the order in which triggers become *true* or *false*. Since the checker has no deep semantic knowledge about the trigger conditions and their correlations and time-dependencies, a full check is impossible.

There are some differences between the reachable contexts that the checker generates and the contexts that are reachable at run time. It is possible that at run time, due to correlation, two triggers will always fire simultaneously, although they are connected to

different rules. Whereas the checking algorithm will generate the intermediate contexts that result from firing each trigger separately, these contexts will never be reached at run time. Only if the triggers refer to the same rule, no intermediate context will be created when the checker simultaneously fires the triggers.

While the analyzer checks for many types of error, only four kinds can occur at run time, and only one is detected and acted upon. These are:

Conflict. The Inference Engine will fire one of the conflicting triggers, depending upon the internal representation which is related to the textual order of the ON statements in the knowledge base. Logically, the Inference Engine acts as if, with triggers t1, t2, t3, etc, *trigger expressions* t1, t2 *and not* t1, t3 *and not* t2 *and not* t1, etc, are formed. Figure 7.12b shows an example: either the middle trigger will fire and the left and the right will not, or the left and right will fire simultaneously and the middle one will not. The second case leads to a non safe state, s3.

True STATE rule becomes true. In Petri net theory, the non-safe state would get two tokens. In SIMPLEXYS, this has no meaning; the Inference Engine merges the tokens.

System cannot stop. This error is not recognized by the Inference Engine. The checker guarantees that in a correct net there is at least one path to the final context, but it does not have the deep domain knowledge to ascertain that such a path will indeed be taken.

Deadlock. A deadlock is the only fatal run time protocol error. The expert system halts and reports the context in which the deadlock occurs. The checker, however, detects all possible deadlocks; therefore these will not occur in a net for which no deadlocks were reported.

7.5. Conclusions

Because SIMPLEXYS has formalized the time related aspects of protocols into a Petri net, it allows consistency checks of the dynamics of those protocols that most other expert system languages and tools do not have. Because the checker has no deep knowledge of the domain, this checking cannot be complete, but the currently implemented checks detect many types of errors and have proven to be very useful.

8. Data and data processing

The expert system's function is to analyze real-world data. But *which* data are to be analyzed, and how they are to be acquired and preprocessed is highly application-dependent. Yet, in real time monitoring applications, the basic signal processing procedures will often be comparable, as they perform the acquisition and processing of a more or less standard set of measurements of a similar character.

Two problems have a general character. The first is that the acquisition rate of many of the measurements is so high that neither a human [Blom and Beneken, 1982; Beneken and Blom, 1983] nor an expert system would be able to handle those 'raw' data; some sort of *algorithmic* data preprocessing (data compaction, feature extraction) is required so that the expert system will be offered more meaningful data at a much lower rate.

The second problem is that the quality of the data is to be suspected [Divers, 1987]. Due to a variety of sources, the acquired data may not reflect the quantity that they are supposed to represent. A process of *data validation* is required to establish the authenticity of the acquired data.

This chapter describes how signal processing might proceed in a patient monitoring system that incorporates an expert system to provide 'intelligence'. The exact nature of the monitoring functions and the intelligence cannot be considered here, since these are highly application-dependent.

We assume that some basic information will always need to be made available to the monitoring system. This basic information will be preprocessed, compacted, validated and stored in a *data base* which can be consulted by the expert system. We therefore describe a 'library' of algorithms that are more or less independent of the final application. Whether they are suitable for an individual application remains to be decided by the system builder, of course. If not, they may provide some useful ideas.

Our earlier research has been called 'servo-anesthesia' [Beneken et al, 1979]; it focussed mainly on automatic data acquisition [Beneken et al, 1983], data validation [Meijler and Beneken, 1987], data processing [Beneken et al, 1978; Blom et al, 1979; Blom et al, 1985] and data display [Meijler, 1986]. The approach that we will take here is an outgrowth of that employed in the design of the Data Acquisition and Display System (DADS) [Meijler, 1986; Meijler and Beneken, 1987]. In particular, we develop a general method to validate quasi-periodic physiological signals. In order to provide a specific example, this general validation method is then applied to a specific signal: the arterial pressure. The ability to validate the arterial pressure signal is a prerequisite for an intelligent blood pressure controller (chapter 9).

Figure 8.1 provides a conceptual model for the processing of data. How the data is acquired depends on the type of the data. The data validation process determines whether

the data are valid or artifactual. If the data contain much redundant information, e.g. in case of a waveform, the feature extraction process extracts all meaningful features from each period of the signal, such as maximum, minimum, average, period duration, etc. Feature extraction is usually combined with validation. If one or more of the extracted features are abnormal, the feature classification process attempts to determine the type of abnormality. The trend analysis process [Beneken et al, 1982; Beneken et al, 1983] classifies and analyzes the dynamics of the data. The history extraction process builds a compact history of the feature over e.g. the last few hours.

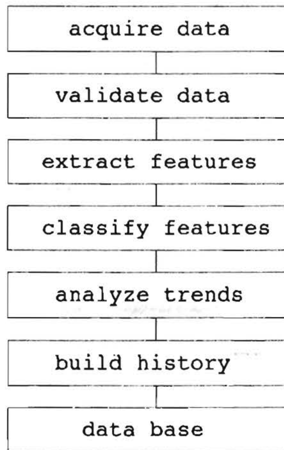


Figure 8.1. Data processing.

The central problem in data processing is the high data rate; new data frequently arrive hundreds of times per second. No expert system is able to analyze the data at such a rate, but neither is this necessary: conclusions are usually not based on these 'raw' primary data but on features extracted from them. Extracting the features from the data is usually an operation that can be performed by *algorithms*, which can be designed to be fast enough to process the primary data. Feature extraction offers three advantages, that simplify the construction of the expert system:

1. the design, implementation and testing of the feature extraction algorithms can proceed independently from the design of the expert system;
2. feature extraction reduces the *quantity* of information to be analyzed by the expert system by eliminating redundancies in the data and/or irrelevant information;
3. feature extraction reduces the *rate* at which the information must be analyzed by the expert system.

The problems that the expert system must solve determine the information that must be available in the expert system's data base. This information in turn determines which signals

must be acquired and which features must be extracted from these signals. The characteristics of the signals in turn determine how the features must be extracted. Because the data processing is so application dependent, it has thus far been impossible to provide general mechanisms that can be incorporated into the expert system tools without compromising efficiency.

An application independent consideration, however, is the *integrity* of the information in the data base. The expert system will often be able to function correctly if data are missing, *not* if it assumes that data are correct if they are not. This makes data validation so important. For a systematic description of data validation as a layered approach see van der Aa [1990].

8.1. Data acquisition

We recognize four different categories of data, that need to be input to the system: continuous measurements (to be monitored at all times), discontinuous measurements (which provide data only once in a while), data volunteered by the medical staff (also infrequently), and demographic data (which need to be entered only once).

8.1.1. Demographic data

Some important non-changing data about the patient are often entered into the system before the start of the operation: the patient's age, length, weight, known allergies, current diseases, etc. These data can be manually entered into the system when it starts up, at which time monitoring is not yet time-critical because the transducers are not yet connected. A much better approach is to have the expert system acquire such data from a 'hospital information system' of some type. Such acquisition, too, must be done when monitoring is not yet time-critical. The FACT rules are meant to store (conclusions about) these data.

8.1.2. Volunteered data

There frequently is important information concerning the progress of the operation that, due to limited functionality of the equipment, cannot be acquired automatically, but which the staff wishes to impart to the system nonetheless, such as:

a. Injections (drug type, dosage).
b. Infusions (drug or fluid type, infusion flow rate).
c. Fluid loss (blood, urine).
d. Settings of ventilator and other equipment, such as gas composition, minute volume, halothane fraction.
e. Interventions. An intervention is a willful occurrence of some event. Many interventions will lead to changes or artifacts in signals.

f. Artifacts. An artifact is a (sometimes willful and necessary) disappearance or disturbance of one of the signals, due to blood sampling, flushing of a catheter, electrocautery, cardiac output determination, transducer disconnection, power line hum, etc.

The *time of occurrence* of these events is essential as well. In DADS [Meijler, 1986] the system assumed a default time of occurrence coinciding with the time the data were entered, but a different time could be specified if it deviated significantly from the time the data were entered.

A practical problem with volunteered data is that, due to the pressures of the task, they are not always entered into the system, or not entered at the proper time. Therefore, in most practical cases, the use of volunteered data should be avoided if at all possible.

Volunteered data are useful to the system only if they can be *interpreted* by the system. If that is not the case, volunteered data can only be stored as a 'comment' that can be entered into the patient's report. Such a possibility should, we think, always be available due to the fact that no protocol can be so perfect that every clinically significant event is anticipated. Automatic deduction of a meaning from such comments by the system would probably demand a very sophisticated natural language understanding user interface, which is currently, and in the foreseeable future, unavailable, at least if we demand that it is both fast and full-proof. Assigning a meaning is best done by having the machine ask the relevant questions. This is treated in the next section.

8.1.3. Discontinuous measurements

Non-continuous measurements are usually initiated by the medical staff and performed only if required. Examples are blood gas and/or blood electrolyte determination, cardiac output determination by a thermodilution or dye dilution technique, and manual blood pressure measurement using an inflatable cuff.

A characteristic of such a measurement is, that the longer ago it was performed, the less its validity as a representation of the present. Old data represent the patient's past, not his present. Each measurement value of this class should therefore have associated with it a time period during which the value is still acceptable, or a function, that serves the same goal, but may take into account other factors, such as the therapeutic state and the variability in some other signals.

There are two ways to acquire discontinuous data. In the first method the user is the initiator. If the machine is to be able to assign a *meaning* to the data, the user may e.g. invoke a menu, then possibly a sub-menu, which presents him with the name of the variable, to which its value is then appended. In the second method the machine is the initiator; it asks a question and waits for the answer. From the point of view of the expert system the difference between the two methods is that in the first case all possible data can be entered in all possible contexts, while in the second case the context determines which data are

relevant, and therefore which questions will be posed; the expert system can compose a *context sensitive menu*. In either case, the expert system must somehow receive the data for use in its knowledge base.

The SIMPLEXYS ASK rule is unable to acquire this type of information, because the system would have to wait for the answer before continuing. In a real time system, however, it should be possible to ask questions without waiting for an answer. SIMPLEXYS does offer the constructs to solve this problem in several ways, e.g. as follows. Four rules play a role for each question[1]. We give an example:

1. rule POSE_QUESTION presents the question;
2. rule ANSWER will receive the answer;
3. rule GET_ANSWER checks whether the answer has been supplied;
4. rule WITHDRAW_QUESTION knows that the question is not relevant anymore and withdraws it if it has not yet been answered.

POSE_QUESTION, a rule of any type, knows that the question needs to be asked and poses it; the question's result (TR, FA or PO) will be returned in MEMO rule ANSWER. The value of ANSWER is set to PO (unknown) pending the answer.

```
POSE_QUESTION: 'pose the question'
...
THEN DO present_question;
THEN PO: ANSWER
```

Procedure present_question shows the question on the screen in an appropriate format. Rule ANSWER will receive the answer; its value will be PO (unknown) until the question is answered. As long as the value of rule ANSWER is PO the rule attempts, in every run, to obtain a value TR or FA through the evaluation of rule GET_ANSWER.

```
ANSWER: 'the answer to the question'
MEMO
IFPO GOAL: GET_ANSWER
```

Rule GET_ANSWER is a TEST rule which continually (each run) tests whether the answer to the question has already been given; it uses the interface function request_reply. If the answer is 'yes', rule ANSWER is set to TR, if the answer is 'no', rule ANSWER is set to FA. As a consequence of either TR or FA, the rule stops its attempts to obtain a value; it also withdraws the question from the screen. An answer 'unknown' (PO) has no effect in this example. Note that the history counter of rule ANSWER provides information about the time the answer was provided.

---

[1] The rule base programmer needs to invent unique names for each rule, of course.

```
GET_ANSWER: 'checking for an answer'
TEST TEST := request_reply
THEN TR: ANSWER
THEN DO withdraw_question
ELSE FA: ANSWER
ELSE DO withdraw_question
```

Rule WITHDRAW_QUESTION is any rule, but most probably a context switch trigger rule. Under the appropriate conditions, it calls Pascal interface procedure withdraw_question which removes the question from the screen if it is still present.

```
WITHDRAW_QUESTION: 'question irrelevant now'
...
THEN DO withdraw_question
```

The above demonstrates that, although the procedures to present and withdraw questions and the function that requests replies are application-dependent[1], such 'real time ASK rules' are easily implemented in SIMPLEXYS.

If the *user* initiates the information transfer, the situation is simpler: the expert system need not maintain the menu; a normal (keyboard interrupt driven) Pascal procedure can do so.

The data rate of non-continuous measurements and volunteered data is normally so low, that an expert system, which analyzes the data in cycles of some 5 seconds or so, will have no trouble handling them.

8.1.4. Continuous measurements

In operating rooms and intensive care units many different physiological signals are monitored and processed in order to assess and quantify the physiological state of the patient. Most of these signals can be temporarily disturbed or invalidated, e.g. by movements, actions of the medical staff or electrocautery. In order to obtain a reliable and correct diagnosis based on (parameters derived from) such signals, the medical staff must be sure that they are valid (not artifactual). This is one of the reasons why the most important physiological signals, such as ECG and arterial pressure, are continuously displayed on monitors.

If signals are to be analyzed by data processing equipment without human supervision, another method of quality control is necessary [Beneken and van der Aa, 1989]. This section describes an investigation into the feasibility of validating signals by the data processing equipment itself, prior to their analysis in a data acquisition and display system. Validation is especially important if the physiological signal is used in some kind of automated therapy,

[1] SIMPLEXYS offers the knowledge base programmer thus every freedom to design his own user interface, including where to position questions on the screen, how to accept answers, etc.

that continues for long periods without human supervision, such as computer based blood pressure controllers or adaptive ventilators.

More and more patient measurements are continuously available and thus must be observed at all times. The vital signs are first processed by the front-end equipment (transducers, amplifiers, filters and such) and then converted, by an analog to digital converter, to numbers that the computer can handle. To faithfully represent the original pulsatile signals, the conversion rate must be fairly high for some signals; the ECG, for instance, may need to be sampled at 200 Hz or more. But there is no need for the expert system to analyze data at this rate: a single sample has no clinical importance; it is the *features* of the signal that are important.

Which signal features are important depends on the signal. In the arterial blood pressure these would be the systolic, diastolic and mean pressure values, the period or rate, and possibly others like the systolic slope or the relative position of the dicrotic notch. It is the task of the so-called *preprocessing algorithms* to extract these features.

There are periods, when the patient's signals are not available: before the transducers are connected, after they are disconnected, during calibrations, etc. During these times no features can be extracted, at least not ones with physiological meaning. A few monitoring devices indicate whether the data that they provide are valid or not; an example is a pulse-oximeter which internally supervises the quality of the signal that it derives its saturation data from. A few other devices indicate that the data that they provide must necessarily be invalid; an example is an ECG-monitor which continuously measures the electrode impedance: a too high impedance leads to an invalid signal, a low enough impedance not necessarily to a valid one. Such information, if it is available, must of course be used by the signal processing algorithms.

Some of the therapeutic equipment may also be able to make its status known to the expert system. A ventilator may have outputs for tidal or minute volume, respiration frequency or even complete flow and pressure curves from which several features can be computed, and from which a change in setting can easily be reconstructed if it does not directly offer such data. In the future we may expect more and more equipment that is remotely controlled from a computer system, with the advantage that not only information presentation, but also equipment control is centralized and that the system is always fully aware of the device's status. An example is a computer controlled infusion pump, that receives its setting as a command from the computer and acknowledges receipt and execution of such a command, in addition to signalling errors (fluid container empty, fluid line blocked, air in line etc.).

In contrast with patient features, equipment features usually do not exhibit spontaneous fluctuations or drift. They are set (and modified) by the medical staff and remain constant over relatively long periods.

8.2. Feature extraction and data validation

Feature extraction is the process which, if the data contain redundant information, extracts from the data only the clinically relevant information.

Data validation is the process which attempts to determine the validity of the data that are entered into the system and that the system needs to derive its conclusions from. Because the therapy will eventually be based on these data, their validity is extremely important. It is always better not to know than to assume incorrect knowledge to be correct. Data validation is usually combined with feature extraction, since it is the *features* that must be validated.

Two steps can always be discriminated in a data validation process. In the first step, available *knowledge about the data* is used to check its correctness. For example, the age of the patient should be between some lower and upper bound, which may depend on the protocol. In the second step, the *mutual consistency* of data is checked, i.e. the knowledge of other data is used to check the current data. For example, a 2-year old patient cannot have a weight of 80 kg; either the age or the weight is incorrect, possibly even both.

Demographic data are somehow entered by the user. The errors can assumed to be data entry or typing errors. If the data are entered through the keyboard, they can often be validated immediately: if detected as incorrect, they can be asked for and entered again. If these data are acquired from a different source such as a hospital information system, correction is usually impossible.

Volunteered data are typically entered through the keyboard as well, and thus can be validated immediately; if detected as incorrect, they can be asked for and entered again.

It may be possible to correct invalid discontinuous measurements if the device that offers them (e.g. an inflatable cuff blood pressure monitor that automatically performs a measurement periodically) can repeat its measurement on a request from the expert system.

Most of the continuously measured signals are periodic by nature; the period is usually the circulatory or respiratory cycle[1]. If the signal is not periodic (e.g. the temperature or the EEG), we can impose an artificial 'period' of a fixed number of seconds to make validation (and averaging) possible.

---

[1] The period does not have a fixed duration, nor is the shape of the signal identical from period to period; small fluctuations are always present. Therefore we prefer to speak of *quasi-periodic* signals.

To validate a signal, we use previously acquired knowledge of the typical appearance of the signal, its possible variations in shape, and the fact that almost always successive periods resemble each other closely[1].

The proposed general method for validation is based on a period by period (beat by beat) determination of a set of significant features of the signal and on a comparison of these 'actual' features with the features of an 'ideal' signal (see section 8.2.1); for more details see Goossens [1986]. These latter features are called the 'model features'; the 'ideal' signal thus serves as a model. Validation is possible only if we can discriminate between an acceptable and an artifactual signal; this presupposes, that we know what an acceptable signal looks like. One must be able to describe it in sufficient detail (by a set of signal descriptors or signal features), but not in so much detail, that an abnormal but valid signal is classified as artifactual.

Thus continuous measurements offer an extra validation test due to the short-time stationarity of the properties or *features* of signals. *Dynamic* information is available to cross-check successive segments of the signal with each other.

The validation process proceeds as follows:

a. acquire one period of the signal;
b. extract the relevant 'features' from that period; these features depend on the nature of the signal;
c. check the feature values with the available knowledge about them; this knowledge is usually a set of lower and upper bounds (which may be context-dependent);
d. cross-check the feature values with each other;
e. cross-check the features with other available information, such as the features of other signals.

Whereas step c is a simple comparison, steps d and e need medical knowledge. However, step d may be eliminated by introducing extra (correlated) features in step b. An example: Features of the arterial blood pressure signal are systolic (maximum) and diastolic (minimum) value. But there is also knowledge about the *difference* between the maximum and minimum values (the pulse pressure). This difference can be added as an extra (correlated) feature.

The choice of the features is determined by standard medical practice. Because a concept 'systolic arterial pressure' exists, the systolic arterial pressure should be chosen as a feature. The existence of the concept signifies both the importance and the approximate constancy of

---

[1] Clinicians also need to validate these signals; therefore these signals are continuously displayed on a monitor. Clinicians, too, use their knowledge of (1) which appearances of the signal are acceptable and (2) that successive periods should look alike.

the feature from one period to the next. The choice of the features is thus guided by the medical vocabulary.

To achieve a cross-check between successive periods, a signal *model* is built; this model represents the set of features of a 'correct', acceptable period. The model is built by averaging the features over a number of most recent validated periods (figure 8.2). Thus the model is *adaptive*: it follows changes in the shape of the signal.

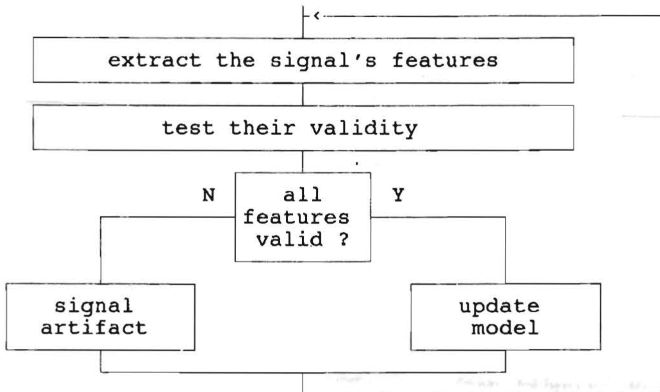


Figure 8.2: Validation of a quasi-periodic signal. In the validation tests the signal's features are compared with both absolute limits and with the model's features. Whenever the discrepancies are too large, they indicate an invalid signal. If all features are valid, the model is updated.

To make validation a practical procedure, we demand some extra properties of the validation process:

- The validation process must be *fast*. Real-time processing of the signal, which is sampled at a relatively high frequency, must be possible.
- In order to give it the required speed, the validation process must be *algorithmic*, not expert system based.
- The validation algorithm must be *compact*. Since many signals may need to be validated, determination of the signal features must not require storage of many samples of the signal.
- The signal model must be adaptive. Signal shapes may change over time.
- When the physiological state of the patient is stable, the signal features must be stable as well.

A suitable validation methodology can be described as follows:

- determine the features to be extracted from the signal; these features will always depend on the *values* of data points and possibly on their *times* of occurrence;
- *segment* the signal on the basis of the features;
- find or design algorithms that can extract these features *without storing samples*; each algorithm will have three functions: *initialize* the extraction, *process* each sample, and *output* the feature's value.

As an example, we will describe the validation of the arterial blood pressure signal. A typical period of the signal is shown in figure 8.3. This figure also shows how the signal is segmented by a set of auxiliary lines at 0%, 25%, 50%, 75% and 100% level (0% approximately corresponds with the previous diastolic pressure, 100% with the previous systolic pressure). Note that the collection of all the points at 0%, 25%, 50%, 75% and 100% is a *model* of the period in the sense that a smooth line through these points provides a fair approximation of the signal. Other signals may require different thresholds (more, fewer, at other levels).

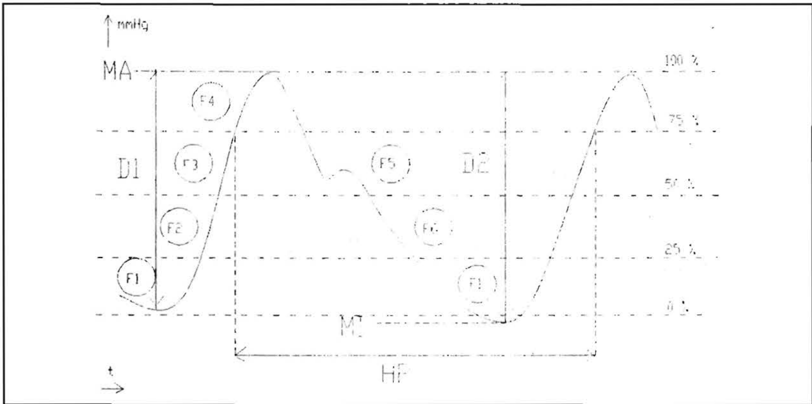


Figure 8.3. A typical period of the arterial blood pressure.

To characterize one period of the arterial pressure, we choose the following features:

- The diastolic pressure *MI*.
- The systolic pressure *MA*.
- The upslope pulse pressure *D1*.
- The downslope pulse pressure *D2*.
- The systolic pressure slope value *H1* between the 25% and 75% levels (a good approximation of the true maximum dP/dT).
- The pulse period *HP* (at the 75% level to provide good noise immunity).
- The average of the blood pressure over a full period *PG*.

The next step is to represent the feature extraction process as a protocol-like structure (figure 8.4). The *states* of the net represent the signal segments between the levels, and they define the way(s) in which each new sample must be processed. The *transitions* represent the crossing of the levels; a transition fires as soon as the sample value reaches a certain level. The net is most often a simple cycle, but this is not necessarily so, because the shape of some signals can change remarkably.

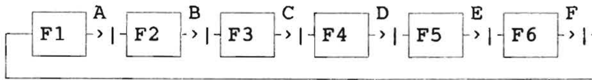


Figure 8.4. A protocol-like representation of the arterial pressure feature extraction process.

In each of the states the sample must be processed, possibly in several ways. In state F2, for instance, we have to update the data for the slope, the mean pressure and the pulse period; each of these updates is done by its own special procedure, which can be executed in several states. The protocol sequencer thus takes the form[1]:

```
repeat
  repeat
    get sample;
    perform state F1 work: update ...
  until transition A;
  repeat
    get sample;
    perform state F2 work: update slope, mean and period data
  until transition B;
  ...
  repeat
    get sample;
    perform state F6 work; update ...
  until transition F;
until end of data processing;
```

where 'transition A' is simply the test 'sample > 25%', etc.

The next step is to connect *procedures* to the states and the transitions; these procedures take care of the basic signal processing operations. As an example, we consider the processing of the diastolic pressure. Going down from segment F5 into F6, we start to look for the absolute minimum of the signal, and going up from segment F2 to F3 we are sure that the minimum has been found; the hysteresis provides for ample noise immunity. Thus the transitions between F5 and F6 and between F2 and F3 are important, as well as the states F6, F1 and F2.

---

[1] This is a slight simplification. Due to sampling, the possibility exists that not even one sample will occur in a segment in which the signal has a very steep slope.

Now that the basic feature extraction algorithm has been described, we need to add a few more details. First, the auxiliary levels can only be determined if we know the systolic and diastolic values. This is simply done as follows: given the knowledge of the maximum duration of a period, sample the signal for that time duration and determine its maximum and minimum value over that time. This is done in the protocol's added state L (figure 8.5), which is left, through transition W, as soon as both minimum and maximum are certain to have been obtained, i.e. after a known fixed time. The only validation that is performed in this phase is a test on the permissibility of the minimum and maximum. This makes the initialization a somewhat vulnerable period. If the initialization is started during a period with many artifacts, some restarts may occur before proper operation commences.

Second, we need to synchronize, i.e. step into the correct context. This is done by adding extra states S and F0 to the protocol, in the following way:

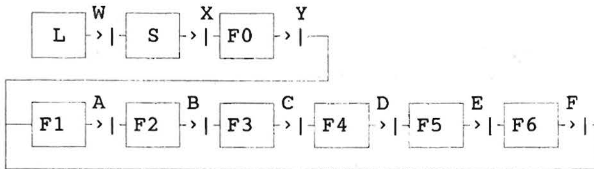


Figure 8.5. A protocol representation of the arterial pressure feature extraction process, including start-up and synchronization states.

State S is the first synchronization state. Transition X is taken as soon as a sample > 75% is found. State F0, the second synchronization state, waits for the signal to go below 25%. We then arrive in state F1. This completes the synchronization.

Synchronization may be lost again if the signal is absent for some time, e.g. in case of artifacts or if the signal level suddenly shifts. In the latter case it is necessary to relearn minimum and maximum values; in all cases it is necessary to regain synchronization. Remaining in a state for too long creates a 'time-out' error. Therefore transitions from state F1 through state F6 to state L must be added to the net; these transitions are to be taken after a specified time, if the normal transition has not taken place. A protocol sequencer step then takes the form:

```
repeat
  get sample;
  perform state's work
until transition or time-out;
if time-out then goto L;
```

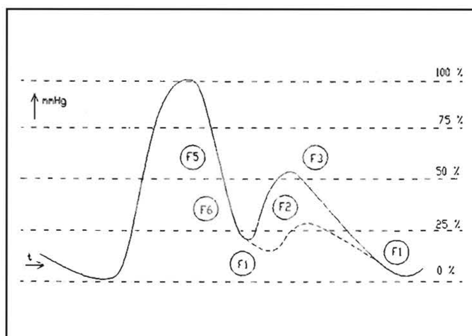


Figure 8.7. An extra peak.

Clinical tests brought to light, that the validation algorithm was not complete yet. Curves with a high extra peak, as shown in figure 8.7, were handled incorrectly. A modification to the protocol, as shown in figure 8.8, corrected this. In the new protocol, it is possible to return from F3 to F2 and from F2 to F1 on downslope conditions.

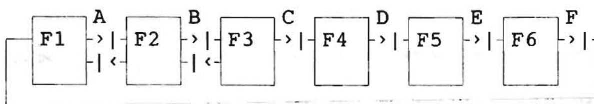


Figure 8.8. The corrected protocol.

At some point in the cycle, all feature extraction procedures have delivered their results for that period. These results can now be validated. The first validation test is against limits, e.g. is the diastolic pressure greater than 20 and less than 200 mmHg? If not, the value is rejected.

For the second test, a *model* of the feature is necessary. This model takes the form of the *average* of the feature over the last few periods. For ease of computation, a *running average* called *model* is computed:

```
model (0) := feature (0);
model (n) := (1-w) * model (n-1) + w * feature (n),
                              with 0 < w < 1 and n = 1, 2, 3, ...
```

Index n is a counter, which is incremented each time a new valid feature value becomes available. The weighting factor w defines a 'time window' for averaging. If $w = 0$, the new feature value has no influence on the model, i.e. the model is not adaptive. If $w = 1$, the old model value is not used in calculating the new model value; this results in a very adaptive model, but one without memory. A compromise must be found between these two extremes, depending upon the maximum possible rate of change of the feature. A value of 0.2 generally works well [Blom et al, 1981], but the value is uncritical: any value between 0.01

and 0.25 may perform well, depending upon the feature's variability. Some testing is required to find a suitable value.

Research must also provide the knowledge about how far the feature's value is allowed to be removed from the model's value for the feature's value to be considered correct. Hoogendoorn [1989] has demonstrated that the difference between feature value and model value usually has an approximately normal distribution, and that interpreting a difference of less than 3 or 4 standard deviations as valid generally detects all artifacts, although a few valid values may be rejected as well [Blom et al, 1981]. That the limits are uncritical was also demonstrated by Zwart [1990], who provides the data about the *differences* between successive pulse periods HP for one patient during an analysis time of 96 minutes that is shown in table 8.1; assuming that all artifactual values can be found in the two highest categories, a satisfactory boundary can be placed anywhere between 3 and 10 (* 20 ms).

diff	total
0	4419
1	3655
2	221
3	33
4	12
5	9
6	3
7	2
8	2
9	1
10-20	16
> 20	90

Table 8.1. The difference between successive pulse periods HP in 20 ms units versus the number of occurrences. Analysis of 96 minutes of data.

Initially, as long as no model is available, only the test against limits applies; this initial phase is called the learning period. As soon as a reliable model is available, both tests are invoked in the artifact detection; other changes are, first, that the model is updated only with *validated* values, so that artifacts do not disturb the model, and second, that the auxiliary levels are determined using the *model's* minimum and maximum values, so that all levels become adaptive as well. In the implementation of the arterial pressure validation algorithm, the limits for every signal parameter were chosen experimentally in such a way that good performance[1] resulted; the values that were used are given in table 8.2.

---

[1] This was done by comparing the algorithm's outcome (valid or invalid period) with a human judgment. In many cases it was very difficult to establish a hard boundary condition, because small perturbations of the curve do not lead to erroneous feature values. The qualification 'good performance' results from the observation, that in recordings obtained during 36 surgical procedures each lasting several hours, no unacceptable feature values were considered valid by the algorithm, i.e. no false positives occurred.

unit	mmHg					ms	mmHg /s
	MA	MI	D1	D2	PG	HP	H1
minimum	40	20	10	10	30	200	50
maximum	270	200	150	150	230	2000	5000
diff	10	10	10	10	10	200	1000

Table 8.2. Acceptability limits for the features (see figure 8.3 for a definition) of the arterial blood pressure. The table should be read as follows (ref. first column): a new maximum (systole) is to be accepted as valid if its value is between 40 and 270 mmHg and if it differs by less than 10 mmHg from the model (average) value.

A validation method will generally be application-dependent; if more features need to be validated, it might need expansion. Two examples will be given for the arterial pressure, neither of which is important in the blood pressure controller application. Figure 8.9 shows that a first minimum can be lower than the diastole; the current algorithm in this case outputs the minimum rather than the diastolic value. Figure 8.10 shows a curve with an extra peak higher than the 75% level; the current algorithm will find two periods rather than one, with quite different feature values. The model will lock onto one of them (the one that occurs most often), and will flag the other one as invalid.

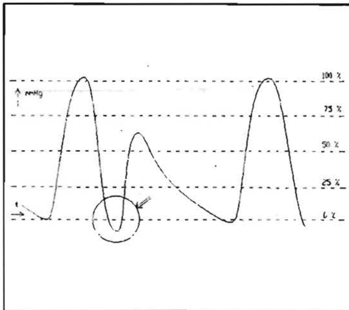


Figure 8.9. A minimum lower than the diastole.

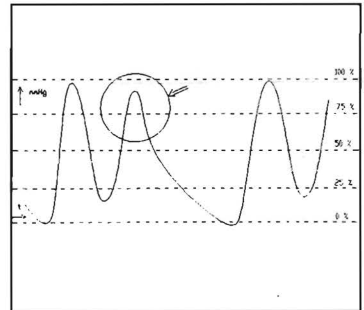


Figure 8.10. An extra peak higher than 75%.

The reason that, in the controller application, the validation performs satisfactorily is that the blood pressure controller does not need a diastolic value and is quite resistant to missing periods.

The validation algorithm is conservative. Figure 8.11 shows a segment of a trend display of a patient's mean arterial pressure; invalid values are not plotted. Generally a few valid[1] periods will be flagged as invalid, but major artifacts will never be considered valid.

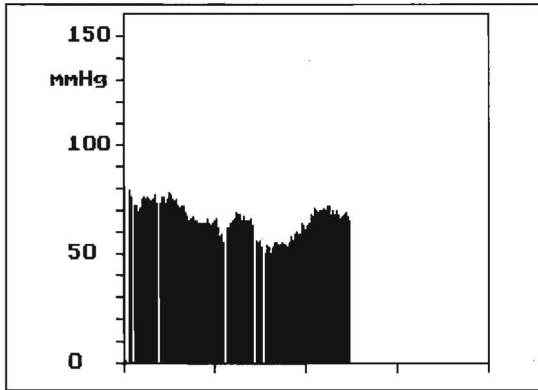


Figure 8.11. The mean arterial pressure: display of invalid values is suppressed.

8.2.1. Classification of invalid periods

The validation algorithm does not know about the *cause* of an invalid period of the signal, but in some cases it may be able to provide the necessary information to further analyze the underlying reason.

In principle, a signal period can be considered valid if all of its features pass these four tests:

test 1a: the value is larger than the absolute low limit;
test 1b: the value is smaller than the absolute high limit;
test 2a: the value is larger than the model's low limit;
test 2b: the value is smaller than the model's high limit.

Thus every period has associated with it a status that describes which of the tests it passed and which it failed. This status can be used to classify abnormal periods.

8.2.2. Limitations of the validation algorithm

The validation algorithm described above is based on the following assumptions:

[1] It remains a problem, however, to find an indisputable distinction between valid and invalid.

1. The signal is quasi-periodic, i.e. all periods of the signal are approximately identical.
2. Each period of the signal can be characterized by a number of features that describe it more or less completely in terms of its clinical information content.
3. Each of these features has known limits, both static (upper and lower value) and dynamic (allowed difference from its 'average' or model value).
4. Each of these features is easy to compute.

In some cases, these assumptions may be inappropriate.

- Assumption 1 is necessary in order to be able to build a model. This assumption is violated if each period of the signal is very different from its neighbors, such as when the patient has a severe dysrhythmia or when a balloon pump supports on every second beat only. Artifact classification may be able to detect dysrhythmic beats, if they do not occur too often. An alternative approach would be to implement several different signal models that operate in parallel and to use the best fitting model for each period.
- Assumption 2 may not be satisfied if the signal has a great number of features, all of which are important. We do not expect this to be the case with the standard range of currently monitored signals except possibly for the spontaneous EEG, where it may be difficult to define 'features'.
- Assumption 3 may be violated in 'uncalibratable' signals, such as the fingertip plethysmogram.
- Assumption 4 may offer a practical limitation. Models for some signals, e.g. the respiratory oxygen concentration [Abkoude, 1981; Coolegem, 1981], are simple. Good models for the capnogram [Rademakers and Schelle, 1983] and the ECG are more complicated than the simple arterial pressure model that is presented here (a more complex arterial pressure model has been reported by Plasman [1981]). For none of the presently monitored physiological signals the computational effort would probably be overwhelming, however.

Some artifacts cannot be detected by the algorithm, even if a perfect model were chosen. A slowly clotting arterial catheter will result in a slowly decreasing systolic-diastolic difference. The same decrease could have a physiological origin, however, in which case it would not be an artifact. More advanced algorithms exist to detect such trends (see section 8.4); moreover, more than one signal may need to be analyzed to uniquely determine the origin of such an occurrence.

8.3. Analysis of features

Frequently an expert system will use a *symbolic* rather than a *numeric* value of a feature. Static feature analysis attaches symbolic values to the features, such as normal, high or very low [Beneken and Gravenstein, 1987]. For each feature, the values shown in figure 8.11 need to be specified, either by the user when limits can somehow be set, by the system from its

knowledge of the patient's state and the protocol of the operation, or from independent (demographic, statistical) information. These limit values may be context-dependent.

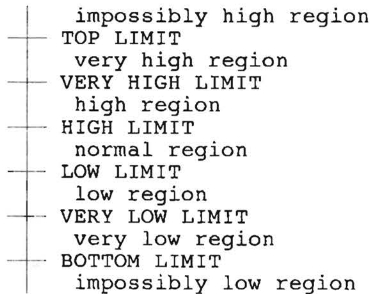


Figure 8.11. Segmentation of feature values into ranges.

The values TOP LIMIT and BOTTOM LIMIT will usually coincide with the absolute limits employed by the validation algorithm, implying that 'impossible' non-physiological values are never offered to the expert system; instead, a 'data invalid' flag will then be available.

If the expert system needs information about the *slope* or *trend* of a signal [Beneken and Gravenstein, 1987], the trend analysis algorithm [de Ruyter, 1982; Peters, 1984; Schoor, 1986; Meijler and Beneken, 1987] is available. It uses a Kalman filter to estimate the slope of a noisy signal; in addition it provides the accuracy of the estimated value of the slope.

A new trend is detected as soon as the slope of the signal significantly deviates from the previous slope; this is indicated by divergence of the Kalman filter. As soon as possible the new slope value will be calculated as well as the time when it started. This time can only be estimated; thus a start time interval is given rather than a start time.

A steep slope, a fast rise or fall, is analyzed further: as soon as possible, it is given a status of either *step* or *impulse*, depending upon the signal behavior after the rise or fall. If the signal stabilizes at a new value, a step is recognized. If the signal returns to the value it had before the fast rise or fall, it is considered an impulse.

8.4. The signal data base

To enable the expert system to provide the best support for the anesthetist, it must be provided with the same information as an anesthetist uses in his management of the patient. In general, this is impossible: the anesthetist derives quite a lot of information from observation of the patient, the equipment and the actions of the personnel, himself included; much of this information is not available to the system because it cannot be acquired automatically and it would take too much time to enter it manually into the system. Besides, algorithms cannot consider the signals in the same ways as the clinician. The signal data base thus has a necessarily restricted character; expert systems can therefore only offer *support*, not replacement of the anesthetist.

8.4.1. Representation of data in the data base

At the moment, it is not yet clear in which way the data are to be represented in the expert system's data base, although some steps to solve this problem have been taken [van der Aa, 1990]. The implementation of a device-independent expert system will be much easier if data are stored in a uniform manner.

On one hand, we would like to have a data base, that is as general as possible. This has the advantage that it is not necessary to create, for each type of operation and each set of equipment, a new data base format. However, such a general data base would contain many entries that never will become valid during an actual operation, either because some equipment is not available in a particular operating room or because some measurements will never (need to) be done. This would not only create a storage problem, but the expert system would continuously need to consider all those dispensable facts and rules, which could prove disastrous in a real-time system.

On the other hand, we could have a data base that is as relevant as possible for the equipment and measurements actually used. This eliminates storage of data that will always be invalid; it eliminates storage of rules that will never return a meaningful result; and it eliminates the time necessary for testing all those superfluous data and rules. But such a data base, optimized for one particular situation, would prevent the construction of a more generally applicable expert system.

The availability of FACT rules offers a partial solution, using the following approach:

- the 'virtual' data base will have some general format; it will contain all measurements and equipment that is or could be normally used;

- the actual data base will, based on knowledge of the actual operating room equipment, the drugs to be used, the measurement devices available in the operating room and such, be composed out of the conceptual data base in such a way, that it reserves no storage locations for data that are known to be unavailable;

- similarly, the 'virtual' expert system rule base will contain all rules that might be needed in any of the operating rooms and types of surgery we consider;

- the 'virtual' rule base will, for each individual case, be compacted in such a way, that tests on data that will never become available during this specific operation in this specific operating room, are eliminated.

In practice, this can be realized by a compilation scan of the expert system rule base, performed before the start of the operation, when the above mentioned facts are known. Even if the compilation scan takes a very long time, it needs never be done on-line. Several

compiled rule bases could co-exist, and only the relevant one could be loaded into the system when it was needed.

8.4.2. Necessary future research

Although data processing seems such an application dependent problem that it cannot be approached in a uniform manner, a more uniform approach may evolve more easily at the level of the representation of the features. In that case, a 'library' of signal processing algorithms could be made available, where each algorithm delivers its output in some uniform manner. But even then, some remaining problems to be solved are:

- Many signals can be measured either continuously or discontinuously, depending on the equipment available, the severity of the patient's condition and the need of the anesthetist. An example is the arterial pressure. Usually continuous measurements can provide more features, and their analysis can be more complete. It would make the expert system much simpler if continuous and non-continuous features had a similar data representation and storage. How to choose a good 'high level' data representation is as yet an unsolved problem.

- In some cases equivalent features can be obtained from several signals. The heart rate, for example, can be derived from the ECG, the arterial pressure waveform or various other sources. In these cases it would be useful to compact all equivalent features into one, with the advantages of simpler expert system software, better artifact elimination and possibly reduced data storage. It would be necessary, however, to check the mutual consistency of the data. Inconsistency could provide extra information about measurement errors and device failure.

9. An intelligent blood pressure controller

During some types of surgical procedure the patient's blood pressure needs to be controlled at a lower than normal value. Usually the drug Sodium NitroPrusside (SNP) is infused to lower the blood pressure. An earlier study [Blom and de Bruijn, 1982] into the dynamic and static characteristics of this drug shows that application of the drug is often difficult due to little *a priori* knowledge of the patient's sensitivity, the change of the patient's sensitivity in time, a pronounced non-linearity, a significant and possibly changing delay time in the control loop and a frequently bad quality of the measured blood pressure signal. Manual control is often difficult and requires close attention to the patient's response. Existing closed loop controllers do not cope well with non-average patients. An automatic controller thus far is not a good alternative for manual control, since those cases which are the most difficult to treat manually are as yet impossible to control automatically. The goal of this research was to realize a robust, unconditionally stable automatic controller.

In many hospitals, the use of SNP infusion for the control of arterial pressure during surgical procedures is common. In most cases, the pressure needs to be reduced from the control value to about 70 mmHg and maintained there typically for 20 to 30 minutes; in cardiac surgery the hypotensive period can last for several hours. After the need for the induced hypotension passes (for instance after clipping an aneurysm), the pressure is allowed to return to control values without overshoot.

The justification for this research is threefold. First, the clinical problem, controlled hypotension, is worth solving, and preliminary investigations have demonstrated the likelihood that this can be done [den Brok, 1986; den Brok and Blom, 1987; Bierens, 1987; Hoogendoorn, 1989]. Second, the incorporation of a real time expert system to embody the knowledge and expertise of the clinician gives us the opportunity to gain more experience in how to use SIMPLEXYS real time expert systems in complex patient monitoring tasks. Third, this study could result in a marketable product, either as a stand-alone system or integrated in an infusion pump, a liquid management system or a work station.

This chapter first describes the basic necessary knowledge about SNP and SNP controllers (section 9.1); more details can be found in Hoogendoorn [1988]. Section 9.2 then describes which knowledge is implemented in the new expert system SNP controller, while section 9.3 gives some implementation details; a complete documentation of the knowledge base can be found in Lammers [1990b]. Section 9.4 describes the major results of the clinical tests; these are more fully described in Zwart [1990]. Section 9.5, finally, formulates some conclusions and gives some recommendations for future research.

9.1. Knowledge acquisition

9.1.1. Controlled hypotension and sodium nitroprusside

Controlled hypotension, also known as induced or deliberate hypotension, is frequently applied during or (shortly) after surgery. Hypotension is a state in which a person's arterial blood pressure is below its normal level, in comparison with hypertension (blood pressure above normal level) and normotension (blood pressure at normal level). Controlled hypotension often facilitates surgery by reducing bleeding but, when practiced erroneously, can cause irreversible damage, including death.

For over 40 years attempts have been made to reduce blood loss by controlled hypotension. Controlled hypotension reduces bleeding into a wound, thereby providing the surgeon with both better visibility and technical freedom for a more definite dissection; this is especially important in excision of malignancies. With less bleeding, the extent of ligated or cauterized tissue is reduced, the chance of infection is minimized and wounds heal better, a prime concern of plastic surgeons. This technique is also used to facilitate delicate surgery of the middle ear and removal of highly vascular tumors, as well as for certain gynecological, urological, orthopaedic, neurological and cardiac operations. Controlled hypotension decreases surgical hemorrhage, permitting a drier field and, should rupture occur, hemorrhage is more easily controlled than under normotensive conditions. Also, by reducing bleeding, the need for blood transfusion during and after surgery is reduced.

Controlled hypotension also has its hazards [Hoogendoorn, 1988]. These potential hazards can generally be avoided by careful monitoring and attention to detail. Most hazards of controlled hypotension arise from lack of experience by the anesthetist and poor communication with the surgeon. Complications also arise when the technique is used on the wrong type of patient (a patient with contra-indications) or in the wrong type of surgical procedure (where other techniques ensure less danger).
Paralyzed patients are unable to respond to pain by movement or phonation; the only signs suggestive of insufficient anesthesia are sweating (often marked following omission of atropine premedication), secretion of tears and a rise in systolic blood pressure. Some blood pressure controllers have reportedly become unstable if during controlled hypotension anesthesia is insufficient.

There is no arbitrary level at which the blood pressure level is optimal. In the young and healthy, the usual level is 60-70 mmHg systolic, whereas in older individuals satisfactory conditions are attained at higher levels. Little is gained by depressing blood pressure more than necessary. Nor is there an absolute limit to the duration of hypotensive anesthesia. The pressure is not kept at a fixed level but is allowed to fluctuate within reason, depending on surgical needs.

Although hypotension can also be caused by other factors such as general anesthesia, body tilt and positive airway pressure, controlled hypotension is best achieved by specialized drugs. Ganglionic blockers like hexamethonium, trimethaphan and pentolinium were once commonly used but now are largely replaced by a vascular smooth muscle relaxant, sodium nitroprusside. This is because ganglionic blockade effects both sympathetic and parasympathetic ganglia, and so additional effects include dilation of the pupils, retention of urine and tachycardia. Sodium nitroprusside increases the internal diameters of the third and fourth order arterioles (15 - 42 μm) while the venules are not affected. The hypotensive effect is due to this increase in the diameters of the arterioles, which corresponds with a decrease of the left ventricle's afterload, the peripheral resistance. For a description of the pharmacology, pharmacokinetics, toxicology and therapeutic indications of SNP see e.g. Pace and Westenskow [1983].

The action of SNP is rapid in onset and of short duration, and therefore is given as a continuous infusion; the effect disappears soon after the infusion is stopped. The body shows a fast reaction (15 to 60 sec) to the infusion, while the full effect takes about 5 minutes to develop. The response time when sodium nitroprusside infusion is stopped is of the same order. Wood et al [1987] have performed clinical studies of the sensitivities to sodium nitroprusside in young and elderly patients to investigate the correlation between age and dose requirements. They found that the sensitivity to sodium nitroprusside increases with advancing years. This increased sensitivity may result from 1) diminished baroreflex activity, or 2) resistance of cardiac adrenergic receptors to catecholamine stimulation, or 3) alteration of direct vasodilation effects. The average patient response observed to sodium nitroprusside is a steady state gain of approximately - 0.3 mmHg/(mg/hr).

To gain a better understanding of the variability of the response to SNP, experiments with 19 Yorkshire pigs were done [Blom and de Bruijn, 1982]. Surgical and anesthetic procedures were standard. The arterial blood pressure was measured in the abdominal aorta and SNP was administered by means of an IMED 929 computer controlled isovolumetric infusion pump. Data were acquired and processed by a PDP11/03 based data acquisition system [Blom et al, 1981a], which also controlled the pump.

Static characteristics were obtained from dose-response measurements. Infusion rates of 0, 1, 2, 4, 6, 8, 10 and 14 μg/kg/min of a standard solution of 100 mg SNP in 400 cc glucose 5% were administered until a new steady state was reached, which usually took 5 to 10 minutes. This procedure was performed both at the beginning and at the end of the experiments, which lasted from 5 to 8 hours. In these experiments, a prototype adaptive blood pressure controller was tested. The conclusion of the tests was that the controller was not sufficiently reliable; the controller attempted to establish the patient's sensitivity from a Kalman filter based correlation between input (SNP flow rate) and output (mean arterial blood pressure), but although frequently a sufficiently accurate sensitivity estimate was obtained, the estimate often proved to be unreliable due to the bad signal-to-noise ratio: the

arterial pressure was often much more influenced by a multitude of other events than by a change of SNP flow rate.

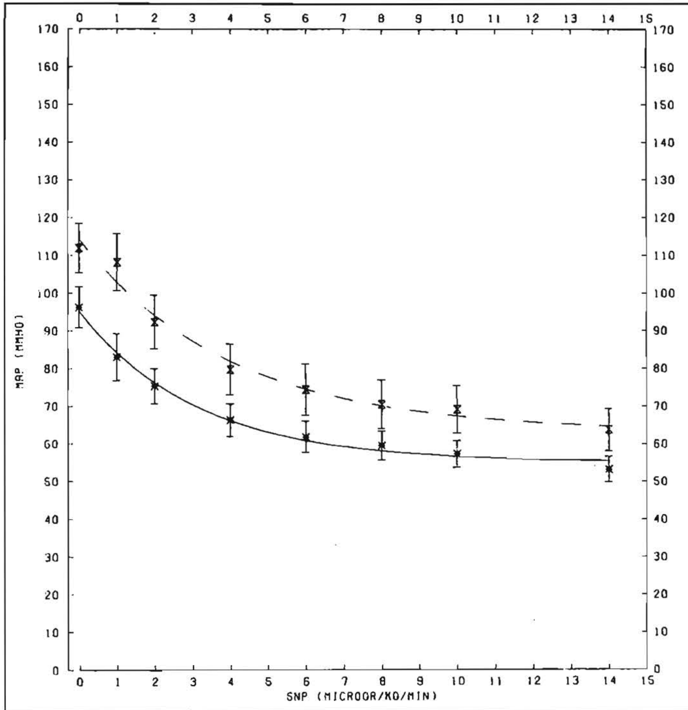


Figure 9.1. Average normalized static characteristic of SNP in pigs. The lower curve was obtained at the beginning of the experiment, the upper curve approximately six hours later.

Dynamic characteristics were obtained continuously by correlating the mean arterial pressure measurements with the SNP infusion flow rate while either of three controllers was in effect: manual control (i.e. the anesthetist specified the infusion rate), an automatic controller designed by Sheppard [1975] and the new adaptive controller prototype. In a clinical environment, estimation of the patient's characteristics is only possible using normal peroperative measurements; no additional tests are allowed to obtain additional information. Assuming a stabilizing controller operating around a given setpoint, a procedure is necesssary to estimate the patient's characteristics from measurements of blood pressure and infusion rate alone. Sheppard [1976] used a pseudo-random binary sequence disturbance of a fixed SNP infusion rate to estimate the patient's impulse response. We decided likewise to estimate the impulse response, but simply used the SNP input as given by the anesthetist or the automatic controller. It was assumed, and shown correct, that normal infusion rate changes by an automatic controller or an observant anesthetist are so frequent that a

'sufficiently exciting input' is presented to the system so that the patient's response can be identified (although not so reliably that it can be used in an adaptive controller). The Extended Kalman Filter [Jazwinski,1970] was used to continuously estimate 30 impulse response points at 15 second intervals, representing a total impulse response time of 7.5 minutes. The estimation procedure was made adaptive, so that changes could be tracked.

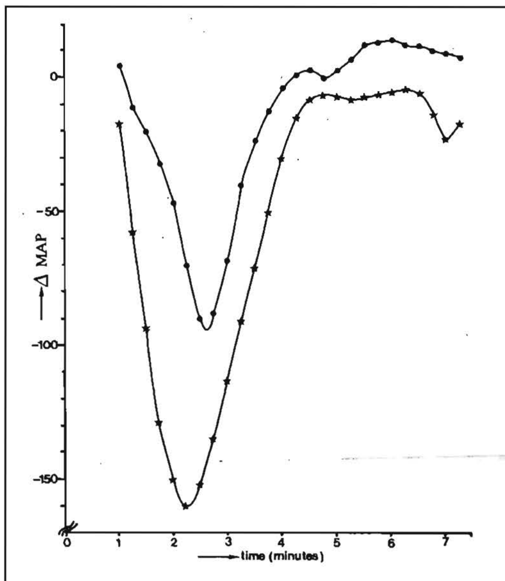


Figure 9.2. Typical dynamic (impulse) characteristic of SNP in pigs. Both curves were obtained at the same mean arterial blood pressure, the upper curve about 90 minutes later than the lower one.

These experiments led to an SNP model consisting of two parts: the static and the dynamic drug characteristics. The static characteristic describes the steady state effect of the drug at different infusion flow rates. The dynamic characteristic describes the time course of the effect when a bolus of one unit is applied (the impulse response). Figures 9.1 and 9.2 show the static and dynamic responses to sodium nitroprusside as determined in the animal experiments.

These characteristics are not constant: at later times different curves were obtained, as shown in the figures. In some of the animals the curves were yet different. Still, those differences were not large when the curves were rescaled: qualitatively all curves were similar. In particular, changes occurred slowly in each individual.

Together, the two curves provide a model. They fully describe the effect of the drug, no matter how the drug is applied. The assumption is, that the same model, possibly with different parameters, is valid for humans as well.

The static curve, shown in figure 9.1, was generally non-linear. In only 11% of the dose-response curves a straight line was a fair approximation, while all curves could be fitted well with an exponential, as follows:

$$x = a + b * \exp(-c * u)$$

where x is the mean arterial pressure, u is the SNP flow rate and a, b, and c are regression coefficients. The three regression coefficients a, b and c that describe each individual curve showed a large variation, however, and could not be related to any of the observed physiological variables (other measured variables, body weight, etc.). The characteristics were not stationary either. In the course of the 5 to 8 hours of the experiments, in 83% of the animals the curve shifted upward significantly. This upward shift showed a large variation, between 8 and 58% of the initial pressure, and could not be related to any of the observed physiological variables.

The dynamic (impulse response) characteristic, shown in figure 9.2, was found to be composed of a delay time of between 15 and 60 seconds followed by a second order response, reaching an extremum after about two minutes followed by a fast decay (see also Slate and Sheppard [1982]). The impulse response never lasted more than 6 minutes. In general, the amplitude of the extremum decreased at lower pressures (which can be explained by the nonlinearity of the static curve) and towards the end of the experiments (which may be due to increasing baroreceptor reflex action). Inter- and intra-animal amplitude (sensitivity) variations of upto a factor of 80 were observed.

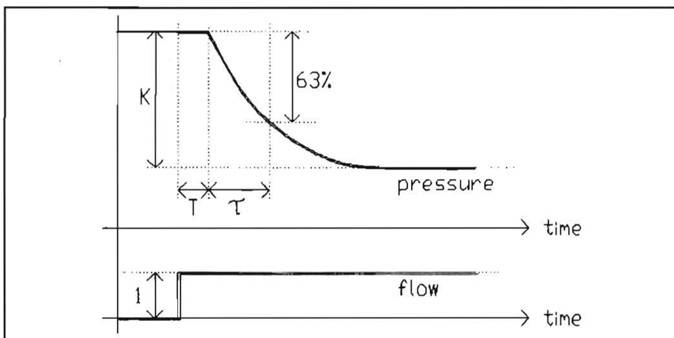


Figure 9.3. The step response.

Important for control purposes is the integral of the impulse response, the step response, which shows the response of the pressure due to a unitary change of the infusion flow rate, as shown in figure 9.3. The complex frequency or s-domain transfer function of the step response can be modelled as:

$$x(s) / u(s) = K * \exp(-s * T) / (1 + s * \tau)$$

where x is the mean arterial pressure, u the SNP infusion flow rate, K the sensitivity parameter (K is negative, indicating that the pressure *decreases* at higher flow rates), T the delay time and τ the dominant time constant (the time it takes, after the delay time, before the pressure has changed 63% of its total response).

Conclusions from this study [Blom and de Bruijn, 1982] were:

1. the dynamic characteristics curve was relatively constant for all subjects at all times; the sensitivity was the only parameter with a broad range: a factor of 80;
2. the static characteristics curve was relatively constant for all subjects at all times; the sensitivity decrease (either as a function of the applied dose or of the pressure) was the only parameter with a broad range;
3. the static characteristics curve showed an upward shift at later times[1].

9.1.2. Sodium Nitroprusside control systems

Although sodium nitroprusside is the most frequently used drug when the arterial blood pressure needs to be temporarily lowered, the use of the drug has its problems. In some patients manual infusion is laborious because the flow rate has to be adjusted frequently; in all patients the infusion's effect must be carefully monitored. Currently and in the past, many researchers [e.g. Slate, 1980; Andre et al, 1985; He et al, 1986; Martin et al, 1987; McInnis and Deng, 1985; Millard et al, 1987; Reid and Kenny, 1987; Rosenfeldt et al, 1986; Stern et al, 1985; Voss et al, 1986; Meline et al, 1986; Westenskow et al, 1987] have designed automatic closed loop controllers. Such a controller has as its task to compute and deliver an appropriate infusion flow rate, so that the measured blood pressure is maintained at some desired lower than normal level (the setpoint). Usually the mean arterial pressure (MAP) is the quantity to be controlled.

This control problem is difficult. The problems are several: frequent artifacts in the blood pressure signal [Rampil, 1987], a large delay time, a pronounced non-linearity of the response, reflexes that try to increase the pressure and that come into action after some time (the combination of the two latter factors results in a system which is not *identifiable* [Cobelli and Romanin-Jacur, 1976; van Genderingen, 1984]), and in general a huge variability of the patient's characteristics, both between patients and for the same patient at different times [Wood et al, 1987; van der Woord, 1981; Blom and de Bruijn, 1982]. The exact pharmacology of SNP is unknown [Coad, 1987; Hutchinson and Hollway, 1985]. SNP is also toxic [Patel et al, 1986; Butler, 1987]; overdosage has to be avoided at all costs. Due to these problems, classical and modern control theory are not good enough [van Genderingen, 1984]. Although not always clearly stated, even modern adaptive controllers won't work correctly all the time [Chizeck, 1986; He et al, 1986; Martin et al, 1987; Millard et al, 1987;

---

[1] This upward shift is assumed to be so slow that no controller will have trouble with compensating for it. From now on, it will therefore be disregarded.

Meline et al, 1986]. Clinical knowledge and experience seem to be necessary for a good control. Many physicians do as well as or better than automatic control systems, at least if they are constantly paying close attention. But that is exactly the problem: they do not; they have more and often more important things to do than adjusting infusion flow rates. Besides, they are prone to errors [Hopking, 1980]. The advantage of computer control may not necessarily be in improved control but rather in the convenience and lack of susceptibility to distraction and fatigue that it provides (computer aided support).

An automated sodium nitroprusside infusion system for blood-pressure control should have certain performance characteristics; some of these have been established by consensus. These include, after a step change in setpoint, a 20 percent settling time[1] of less than 10 minutes, a maximum overshoot of less than 10 mmHg, and a steady-state error within + 5 mmHg. Along with these performance characteristics, the controller also has some clinical constraints. The first of these is a maximum allowable infusion rate, a function of patient weight and drug concentration, to prevent cyanide toxicity. Another constraint is that incremental increases in infusion rate should be limited to prevent rapid decreases in pressure which can cause diminished blood flow or circulatory collapse.

The controller must also be able to handle a wide variety of patient types and a wide range of patient characteristics; the largest number found in the literature concerning the patient's sensitivity variability, for instance, is 48-fold [McNally et al, 1977], while we found a factor of 80 in animals (section 9.1.1). Furthermore, the patient's characteristics can change during the course of the operation. Thus, the controller must be able to identify the type of patient it is controlling and then adapt to any changes that might occur. The performance of the controller must be guaranteed, even during episodes in which no valid measurements are available.

Another complication is that large 'noise' levels may occur in the blood pressure signal. Especially in the ICU, this noise may be due to sudden changes in the patient's emotional state or level of activity in which case the controller must take appropriate steps to regulate the blood pressure. Blood pressure is not always stable in the operating room either. Surgical stimulation during light anesthesia, bleeding, rapid fluid infusion, administration of other vasoactive drugs, and respiration maneuvers can dramatically influence blood flow and arterial blood pressure. However, routine procedures such as the taking of blood samples or the flushing of a catheter give readings which are not real and must be disregarded.

In view of the problems of conventional modeling and estimation methods for this particular problem, we decided to have a better look at the (rather successful) clinical practice of ad hoc adjusting the infusion flow and to investigate the use of a simple but

---

[1] The time required to bring the pressure toward the setpoint, within a margin of 20% of the setpoint change.

robust controller combined with an expert system to monitor the controller's behavior and adjust its parameters if necessary. Robustness is the key issue.

The patient's behavior can be modelled in several ways: Sheppard [1976] gives a Laplace-domain transfer function; several authors [Arnsparger, 1983; Stern et al, 1985] use an ARMAX model. These authors have the following in common:

- They use a model with a fixed structure representing the patient, with parameters tuned for an average patient. By making the controller adaptive (usually the patient's characteristics are estimated by some kind of least squares method) the controller is adjusted for non-average patients and the time variance is taken care of.

- The controllers generally work acceptably for average patients, but for patients who are exceptionally sensitive or insensitive the controllers experience problems with their stability and accuracy.

Our own experience shows that the blood pressure signal is contaminated with much 'noise' due to spontaneous fluctuations and transient disturbances, making it unlikely that a parameter estimation method will consistently produce reliable estimates. Moreover, the characteristics of the system (specifically its time-varying non-linearity and the reflex mechanisms that elevate the 'zero flow' pressure) result in the fact that the system is not (or badly) observable [van Genderingen, 1984], making attempts to accurately estimate the system's parameters almost hopeless.

9.1.3. Proportional Integral Derivative controllers

The basic formula for a PID controller is:

$$u_k = P * e_k + I * f_k + D * g_k$$

where

$k = 1, 2, 3, \ldots$ is the sample moment;
u_k is the newly computed output of the controller;
$e_k = x_k - r_k$ is the offset, the difference between the controlled system's output x_k and setpoint or reference r_k;
$f_k = f_{k-1} + e_k$ is the integral of the difference;
$g_k = e_{k-1} - e_k$ is the derivative of the difference

and P, I and D are constant or variable gain factors. The P-term's function is to bring the measured output to the setpoint. Because P must be finite, the P-term alone cannot achieve this; a constant non-zero offset would be the result. The I-term provides a memory function which takes care that on average this offset is zero. The function of the D-term is to prevent

too large in- or decreases of u_k; it thus acts as a safety mechanism. The D-term is often eliminated, however, because of its adverse influence on stability due to non-specific transients of x_k ('noise' due to a variety of sources). This results in a slightly lower quality of control, but this is compensated by a greater robustness.

The traditional approach in biomedical applications has been to use a fixed-design PID controller to adjust the rate of drug infusion to the controlled system. These controllers are relatively robust, simple to implement, they do not require an elaborate model of the system, they produce zero steady-state error and when combined with logical rules to avoid overdosage, they can be successfully used for clinical therapy. But these controllers also possess several disadvantages. These include their tendency toward instability (for large I) and excessive noise sensitivity (when a large D is used in a noisy environment). There is also no systematic way to alter the controller if the system changes in a real-time environment.

Fixed PID controllers are thus too simple to adequately control such a complex variable as the human blood pressure, because no general method exists to select the relative weights of the three terms when the system description is unknown. Since physiological systems are often poorly characterized and may change with time, it is desirable to use controllers that automatically adapt their operation to changes in the system characteristics. PID controllers can be *tuned* to the system they control if (some of) the system's characteristics are known.

Tuning rules for PID-controllers are well known [e.g. Krijgsman et al, 1989]. In particular, it is recommended to use a sampling interval which is at least a factor 4 to 15 times smaller than the system's settling time[1]. Tuning can be based on the system's performance after a step input or on induced oscillations. Large enough step changes of the input are sometimes available and may lead to an estimate of the system's delay time, gain and integration time constant. Errors in these estimates are likely, due to the bad signal quality of the pressure signal. Moreover, these parameters may change in time, and the estimates may therefore only be useful for a short time. Inducing oscillations cannot be used clinically; unwanted oscillations which should not normally occur might be used, however.

During cardiac surgery so many other factors besides SNP influence the blood pressure that even *a posteriori* estimation of the system's parameters from recordings is quite difficult, even by an attentive and resourceful human observer [Zwart, 1990].

9.2. Knowledge implementation

This section describes the requirements that we determined to be the basis for the new expert system based SNP blood pressure controller, which was to be employed during cardiac surgery. This section states the assumptions from which the controller's knowledge base is derived, as well as some design criteria.

---

[1] The settling time is the time it takes, after an input change, until 90% of the total output change is observed.

9.2.1. The arterial pressure signal

The blood pressure signal, that is output by the monitor, can vary between 0 and 300 mmHg; the normal physiological range is more limited. The measurement noise, the variability in the blood pressure samples due to the measurement process (transducer and amplifier) is small (smaller than 1 mmHg) and assumed to be negligible. Transducer drift is assumed to be negligible. In practice, the transducer is zeroed once in a while. During this process the signal is invalid (and needs to be detected as such).

The blood pressure is influenced by a multitude of other physiological processes; an example is breathing. This causes multi-frequency, more or less random, fluctuations, whose amplitude is assumed to be small (less than 10 mmHg peak-to-peak). These 'normal' variations must not disturb the controller's actions.

Once in a while a large *transient* can be observed in the mean arterial pressure. A transient is a sudden large in- or decrease of the pressure which, even if not acted upon, disappears after a short time. However, when a sudden large in- or decrease of the pressure appears, the control system has no way to 'look ahead' to establish whether it will be a transient or not. It is most prudent to temporarily assume the worst case situation.

Positive transients can be tolerated if they do not last too long. These positive transients are often caused by pain stimuli. Insufficient pain suppression may cause large increases in blood pressure that last as long as the pain stimulus occurs and subside when the pain stimulus is over or when pain suppression medication is increased. Such a blood pressure increase should not be suppressed by SNP, because SNP is not a pain killer. If it were suppressed by SNP, a large pressure undershoot might occur after the pain ends. Since too low a pressure is more dangerous than too high a pressure, a control strategy of, at least temporarily, not responding to a large sudden pressure increase seems best.

Negative transients are dangerous; they can indicate a state of shock[1]. SNP infusion should stop immediately and should only be resumed when the blood pressure is approximately at the setpoint again. However, if the negative transient was due to an artifact of some sort, the expected after-effect of temporarily stopping the SNP infusion will often be a pressure overshoot.

The above mentioned *interpretations* of what causes transients, i.e. pain and shock, will often be incorrect, as transients may have other causes. The control strategy that is followed after such transients was chosen for the sake of patient safety and appears reasonable also if the cause should be different. Anesthesiologists act in a similar manner.

---

[1] This interpretation is probably more appropriate in the intensive care unit than during bypass surgery. Yet the appropriate control decision is the same.

Transients needs to be detected for more reasons. The controller simply cannot compensate for transients: they are too fast and too large. Moreover, an attempt to compensate for a transient would cause a large change of flow, but without much effect. And when the transient is over, the controller must recover and bring the flow back to the pre-transient level. It is therefore better to detect transients and select a different control regime until the transient is over. It may also be undesirable to compensate for transients; rapidly changing blood pressures are a diagnostic aid for the anaesthetist and should not be obscured.

Many disturbances can cause the signal not to reflect the true blood pressure: blood clotting, air bubbles in the line, flushing the arterial line, sampling of blood, electrocautery etc. If these disturbances can cause an incorrect computation of the mean pressure, they should be detected by the validation algorithm, as well as transducer, catheter and amplifier failure. Other disturbances reflect the real blood pressure, but are due to instabilities due to e.g. surgical events or administration of other drugs with a vasoactive effect.

The signal validation algorithm (section 8.2) computes, for each heart period, the mean arterial blood pressure value and sets a validity flag, which indicates whether this value reflects the real mean pressure or not. If not, the value cannot be used. The signal validation algorithm is based on the assumption that almost identical blood pressure periods follow each other; this excludes use of the controller during certain therapeutic procedures (e.g. when employing a balloon pump that supports only one beat out of every two or three) and for some patients (e.g. when the patient's heart rhythm shows persistent bigemini).

The controller has a cycle time of 5 seconds. An algorithm averages the output of the validation algorithm over a 5 second interval; values which are flagged as invalid are, of course, not included in the average. The number of periods included in the average also depends on the heart rate. If in the 5 second interval no valid beats were found, a 'pressure not valid' flag is set by the algorithm. Thus the expert system controller is provided with a new input every 5 seconds, consisting of a MAP value and a validity flag.

9.2.2. The controller

A diagram of the blood pressure control system is shown in figure 9.4. The system communicates with the outside world in three ways: it acquires the analog arterial blood pressure signal from the AD-converter, it communicates with the infusion pump through an RS-232 interface, and it communicates with the user through the computer's keyboard and display.

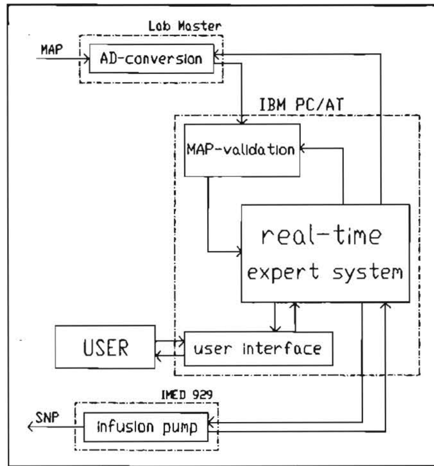


Figure 9.4. The blood pressure control system.

The controller has two control modes: manual and automatic. The controller starts up in manual mode with a zero flow rate and a setpoint equal to the MAP at this time. Through the keyboard, the setpoint can be increased or decreased at all times, in steps of 1 mmHg, between limits. The new setpoint depends on the number of times the setpoint up/down keys are pressed (the keys have auto-repeat); the new setpoint is, in order to be able to detect large setpoint changes, effectuated after waiting for a fixed time (8 seconds) after either of these keys was last pressed. In manual mode, the infusion flow rate can be increased (up to the maximum value allowed) or decreased (down to zero flow) in steps of 0.1 ml/hr (approximately 0.02 μg/kg/min for an 80 kg patient); the new flow rate takes effect immediately (in the next run, i.e. within at most 5 seconds). If the system is in manual mode, it can be switched to automatic. Automatic mode starts up with the flow rate that was set in manual control.

A safety key allows stopping the infusion immediately (i.e. within 5 seconds). When in manual mode, the flow rate is set to zero. When in automatic mode, manual mode is entered with a zero flow rate.

Under certain conditions (air in line; occlusion; low fluid; malfunction), the infusion pump will stop delivering fluid, give an alarm, and will not obey further commands till the problem is resolved. The controlling system needs to know these facts and act appropriately. Handling pump errors is done as follows: if in automatic mode, the system first switches to manual mode; next, the fact that the actually delivered flow rate is zero and not any more under control of the system is recognized, and lastly the appropriate alarm message is displayed. After repair of the alarm condition, which may take any time interval, control can be resumed.

Due to instabilities in the blood pressure signal caused by all kinds of artifacts, the signal can be invalid for periods (much) longer than 5 seconds. Feedback control is then impossible and feedforward (open loop) control is, for safety reasons, allowed for only a short time. The following 'hold mode' solution is chosen for the controller:

- control continues using a MAP equal to the last valid value if the signal is lost during at most 30 seconds;
- if the last valid pressure measurement was less than 20 mmHg from the setpoint, then control continues using a MAP equal to the last valid value during a signal loss of at most 60 seconds;
- if the signal is lost for a longer time, the system gives an audible alarm and control returns to manual mode with a flow rate identical to the last valid flow rate given.

To resume automatic mode, the signal must be valid again; one key-press is then sufficient.

9.2.3. The SNP characteristics

Sensitivity varies widely, both between patients and in an individual patient over time. To ensure safety, the worst case sensitivity range is assumed to be a factor 81, consistent with our findings in animals and higher than the largest factor the literature mentions for humans (section 9.1.3). No simple controller can handle this wide range. In the animals (Yorkshire pigs, see section 9.1.1), normal infusion flow rates varied from 0.1 to 10 micrograms per kg body weight per minute; in humans 0.5 to 5.0 µg/kg/min is usually adequate, with the lowest doses required in the presence of powerful inhalation anesthetics.

As blood pressure decreases, the renin-angiotensin reflex comes into action to fight the lower pressure. Thus, at lower pressures, the patient's sensitivity is decreased. In some patients this effect does not occur. In others, the effect is so strong, that almost no SNP dose can force the blood pressure lower than a certain limit. Reflex action also increases the zero-flow-pressure, i.e. the pressure that would develop after the infusion stops.

9.2.4. The PID controller

Because a PID-controller is simple and robust, it was chosen as the basic controller. Initial tests during 33 cases provided a verification of the animal data about the dynamic SNP response in patients; these are given in table 9.1, together with the ranges that the controller was designed to handle. The controller's nominal gain is - 0.2 mmHg/(mg/hr), approximately equivalent to - 0.04 mmHg/(µg/kg/min); the relative gain varies between 1/9 to 9 in steps of a factor 3, and the control gain is equal to nominal gain times relative gain.

Simulations were then performed to establish the best (most robust) values for the control parameters. This is a partly intuitive process called 'tuning', in which many

qualitative and quantitative compromises have to be made, e.g. how to obtain as fast a settling time as possible while still ensuring a minimal overshoot for a worst case combination of patient parameters. For numerical data on what is meant by terms like 'good' and 'reasonable' see Lammers [1990b].

	minimum found	maximum found	design optimum	design range	
relative gain	0.6	4.0	1	.1-9.0	
time delay	15	75	50	25-100	seconds
time constant	30	100	60	30-120	seconds

Table 9.1. Dynamic SNP parameters. Extremes from 33 cases and the ranges that the controller was designed to handle.

Simulations showed that the derivative term should be set to zero, and that a constant integrative term could be chosen without a significant degradation of performance. The resulting controller is, however, not yet robust enough to handle a sensitivity range of an assumed factor of 81. Simulations established that the controller's parameters could be chosen in such a way that if any single control parameter was a factor 2 off from the counterpart patient parameter, control would still give good performance, while a factor 3 off would still give reasonable performance. The controller's performance would also be still acceptable if the estimates of *two* control parameters were both a factor 2 off simultaneously. Tuning is made difficult by the fact that the performance ranges are not symmetric; if the patient's time delay is too small and his time constant too large compared to the controller's assumptions, this has modest consequences for the controller's performance, whereas too large a time delay and too small a time constant have serious consequences.

Table 9.1 shows that the controller's time constant and time delay can be fixed at an optimum value, but also that its gain needs to be adapted. As a basis for this adaptation, the patient's sensitivity as well as the controller's relative gain are classified into one of five categories, as shown in table 9.2, and a number of 'gain up' and 'gain down' mechanisms is introduced which monitor the controller's performance in various ways and can adjust the control gain in steps of a factor 3.

patient's sensitivity	relative control gain
very insensitive	9
insensitive	3
normal	1
sensitive	1/3
very sensitive	1/9

Table 9.2. Sensitivity and gain ranges.

For safety reasons, the system starts to control at a very low gain (1/9), as it initially assumes a very sensitive patient. In many cases, however, this control gain will be too low. Optimally, the controller's gain should be inversely related to the patient's sensitivity.

A PID-controller's performance can be established from observations in two ways (for details on how the qualifications are quantified for the SNP controller see Lammers [1990b]):

1. In a steady state condition (the mean pressure is or should be approximately constant), the controller's gain is too low if the offset (the difference between the actually measured pressure and the pressure setpoint) is, on average, significantly different from zero for too long a time; and the controller's gain is too high if the offset is, on average, zero, but shows oscillatory behavior.

2. Under dynamic conditions (the mean pressure changes or should change, e.g. due to a setpoint modification), the controller's gain is too low if the pressure changes too slowly; and the controller's gain is too high if the pressure changes too rapidly.

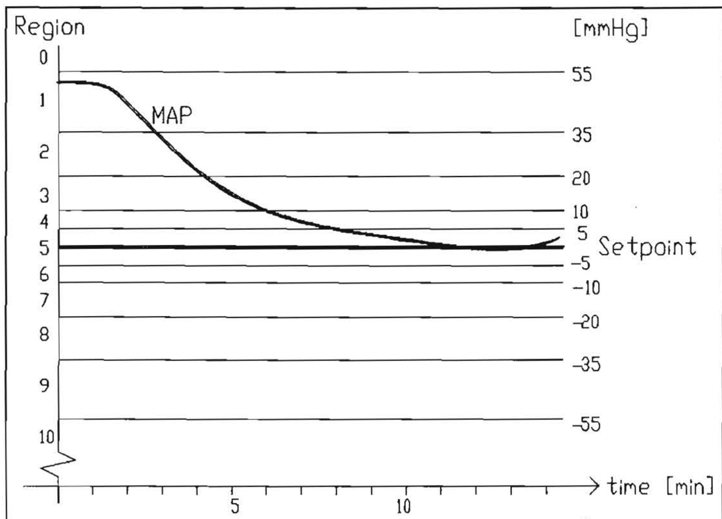


Figure 9.5. Offset regions around the setpoint.

A measure of the MAP progress after a change of the setpoint is thus necessary. If the gain is correct, the progress will be within certain known limits; if the gain is too low, the progress is slower than a limit value and if the gain is too high it is faster than another limit value. The progress of the pressure is monitored by constructing a number of levels around the setpoint (figure 9.5) and observing how long the actually measured pressure stays within

a *region* between two adjacent levels. After a setpoint change, the MAP moves to one of the regions, and control actions will attempt to move it back to region 5.

The width of the different regions is chosen in such a way that if the control gain is correct, the MAP stays in each region for about 2 minutes. If the MAP stays in a region for longer than 4 minutes, the progress is too slow and the relative control gain is tripled, i.e. the patient's sensitivity is assumed to be one class lower (see table 9.2); if it stays in that region for an additional 3 minutes, the control gain is tripled again. On the other hand, if the MAP moves through a region within a period of one minute, the gain is too high and is divided by 3. Progress is thus determined by the time the MAP stays in a region, or rather by a *slope* (MAP change per time period). Direct computation of a slope in a noisy signal is not reliable; to reduce noise influences, the slope should be computed either over a certain time or over a certain MAP change. The adaptation mechanism uses both: if the MAP changes slowly, the gain will be increased when after a certain time the MAP is still found to be in the same region; if the MAP changes quickly, the gain will be decreased as soon as a border is crossed.

To prevent the MAP from oscillating back and forth between regions due to small spontaneous pressure variations, the region borders have hysteresis: the MAP is not considered to have left a region before it is already 2 mmHg beyond the region's border. Figure 9.5 shows, for example, that the border between regions 5 and 4 is at 5 mmHg. The MAP must then be 7 mmHg from the setpoint before region 5 is left and region 4 entered, and the offset must decrease to 3 mmHg again before region 5 is returned to.

Simulations demonstrated, however, that the region-based gain down mechanism alone is not always adequate. Due to the SNP characteristics it takes about 2 minutes before the MAP shows a significant change after a setpoint change, then it takes about one minute before a region is traversed; only then is the control gain decreased, and a significant effect of the latter takes another 2 minutes. In all those 5 minutes, the MAP has moved towards the setpoint and may already have passed it. A faster gain down mechanism, activated by a *persistent large relative flow change*, is added to prevent such an overshoot. This is done as follows. One of the system's safety features is that in each 5 second sample/control period the SNP flow in- or decrement is limited to 7% of the current value (or to 0.07 ml/hr if the flow becomes lower than 1 ml/hr). This means that it takes about one minute to double (or halve) a significant control signal. Because the most effective method to limit the change of flow is to decrease the gain, the gain is decreased if the flow change remains at its maximally allowed value for longer than 15 seconds. This gain down procedure is disabled, however, during the first 30 seconds after a gain up procedure and during the first 15 seconds following a setpoint change.

A third gain down mechanism is active when oscillation is detected. This mechanism is slow, since it must detect several consecutive border crossings; it takes approximately 10 minutes before oscillation is detected. By definition, this is too late: oscillation should be

prevented, not detected. In practice, this oscillation detection gain down mechanism is expected to come into action only rarely; it exists for safety reasons. Some patients are inherently unstable, however; they spontaneously show large MAP fluctuations comparable to oscillations. The best a controller can do in such cases is to decrease the gain, thus stabilizing the control signal, the same action as is required after the detection of oscillation.

In case of oscillation, we expect the *average* MAP to be near the setpoint, in region 5. To detect oscillation, the number of crossings from region 5 to region 4 and from region 5 to region 6 is counted; an oscillation is detected as soon as the counter reaches a value of 4. To prevent a few crossings over a longer time from resulting in an erroneous detection of oscillation, the counter is decremented every 4 minutes (but not below 0). Simulations have shown this to be an effective procedure.

Due to the non-linearity of the SNP dose-response curve, a gain decrease is necessary to ensure safety if a large setpoint *increase* is ordered, since this might mean a move to a region of higher sensitivity. This mechanism acts as soon as a setpoint increase of at least 40 mmHg is ordered within a short time (but only if the flow is currently less than half its maximally allowed value). It *anticipates*: it becomes active before the MAP shows any effects due to the setpoint change. In some cases the gain down action will be inappropriate, and then a gain up action may reverse its effect after some time.

The control formula which is used in the basic controller is written in such a way, that a flow *increment* is computed:

$$u_k = u_{k-1} + G * [I * (r_k - y_k) + P * (y_k - y_{k-1})]$$

where

y	mean arterial pressure	[mmHg]
r	setpoint pressure	[mmHg]
u	flow rate	[ml/hr] = [mg/hr]
G	adaptive control gain	(1/9, 1/3, 1, 3, 9)
I	I-parameter, 0.0960 to 0.0720	[(ml/hr)/mmHg]
P	P-parameter, 0.0056 to 0.0036	[(ml/hr)/mmHg]

Moreover, the flow increment $u_k - u_{k-1}$ is limited to 7% of u_{k-1}, as mentioned above. In the formula, u_k and y_k are the *current*, u_{k-1} and y_{k-1} the *previous* SNP flow and pressure respectively, while the sample interval is 5 seconds.

The P- and I-parameters are allowed to vary, because simulations demonstrated that in regulation different values were appropriate than in stabilization (a phase plane approach is described in Krijgsman et al [1989] to realize a similar strategy). If the offset is 100 mmHg or more, the lowest values are used; at smaller offsets, the parameters proportionally grow to their highest values. For instance, if the offset is 50 mmHg, the actual P- and I-parameters are halfway between the two values given above. This makes the controller non-linear; since the offset will generally be less than 50 mmHg, the non-linearity is usually small.

9.2.5. Safety features

Cyanide is produced when SNP is metabolized [Patel et al, 1986]. Blood thiocyanate is an intermediate product; thiocyanate levels above 150-200 mg/l exceed the detoxifying powers of the body, and acute anoxia may result. This action is reversible, but only if recognized early. Cyanide toxicity should be suspected in any patient who requires a flow rate greater than 10 μg/kg/min; recommended 'safe' infusion rates are less than 2.0 μg/kg/min. This sets an upper limit to the average SNP flow rate. Another reason to limit the flow rate is that if, due to some unexpected occurrence, the infusion flow rate ever reaches a very high value, the controller will not be able to immediately reduce it to the correct level. For both reasons an upper limit must be set for the flow rate.

Cyanide may also accumulate; toxic blood levels may occur if more than 1 mg/kg is given over a period of two or three hours (rates greater than 4 μg/kg/min for more than 2 or 3 hours may be unsafe). The 'safe' maximum dosage has been reported by others to lie between 3.0 and 3.5 mg/kg. This sets an upper limit to the total SNP dose allowed. The method is sometimes abandoned because the patient is considered resistant, if a satisfactory low pressure does not follow infusion of 50 mg over 30 minutes; this occurs mainly in young patients.

There are two types of insufficient response or non-response in which the controller cannot cope anymore: a *low* setpoint cannot be reached despite the maximum infusion flow rate, and a *high* setpoint cannot be reached despite a zero flow. Non-response can be detected in several situations: the pressure stays outside region 5 for a very long time; the maximum or minimum flow rate has been infused for an extended period of time; or the system requests a gain up when the gain is already at its maximum value.

In the first situation the MAP approaches the setpoint very slowly or not at all. If the setpoint is below the present MAP, such a low MAP may not be reachable for this particular patient. If the setpoint is above the current MAP, and the flow is small or zero, the MAP will not rise toward the setpoint. As soon as the MAP is in any region but region 5 for 250 seconds, the gain is tripled. If it stays in that region for another 180 seconds, the gain is tripled again. If the MAP still stays in that region after the second gain adaptation, this is evidence of non-response. This is not certain, however; for less sensitive patients (1/3 and 1/9 class) the gain must be adapted 3 or 4 times before it is correct. Thus non-response is considered to exist only if the pressure remains in a region (any region but 5) for a very long time and if the gain has already been enlarged to at least 3.

The second situation that gives evidence for non-response is that more than 95% or less than 2% of the maximum flow rate is used for a continuous period of more than one minute. To allow the flow rate to temporarily be at its maximum or minimum value after a setpoint change before gain adaptation occurs, we also require the setpoint to be constant for one minute; further setpoint changes will be serviced after this period is over.

The third situation for non-response is a gain up request while the gain is already at its maximum.

If non-response is detected, further gain increases are disabled and the flow rate is limited to its maximum or minimum value. A message is issued only if the non-response is caused by a flow rate larger than 95% of its maximally allowed value.

In the experimental stage of the blood pressure controller it was decided to limit the infusion flow rate to a 'safe' 2 µg/kg/min, a factor 2.5 to 4 lower than the maximum rate mentioned by others [Hammond et al, 1979; Slate, 1980; Packer et al, 1987], but in agreement with standard practice (much used manual settings of the SNP flow rate that we observed were 0.2, 0.4, 0.8, 1.6 and only occasionally 3.2 µg/kg/min; intermediate values were seldomly used). It is therefore to be expected that the controller will not be able to sufficiently depress the pressure in less sensitive patients.

9.3. Design of an expert system based SNP controller

Conceptually, the new SNP blood pressure control system consists of the following sub-systems, that will be described in this section:

1. A data acquisition and validation module (section 9.3.1).
2. A controller module (section 9.3.2).
3. A supervisor module (section 9.3.3).
4. A user interface module (section 9.3.4).

Section 9.3.5 describes some of the knowledge engineering properties of the system which make the system's performance accessible for later evaluation, but which are essentially 'hidden' for the clinicians.

9.3.1. Data acquisition and validation

All timing in the system [Ream et al, 1987] is derived from the sampling process. Interrupt routine *readADbuffer*, installed by the expert system when it starts up, samples the arterial pressure signal at 50 Hz and buffers it (figure 9.6). After calibration of the signal [Nelson and Ream, 1987] is performed, the validation algorithm acquires its data from this buffer, validates the signal and derives the 5 second mean. The inference engine waits till the 5 second mean is available and then processes it. The expert system thus has a cycle time of 5 seconds: after the acquisition and processing of 250 samples the knowledge base is re-evaluated and the SNP flow rate, if necessary, readjusted.

Data acquisition and validation of continuous signals has been described in section 8.2, where the arterial blood pressure signal was used as an example. Although many clinicians are used to controlling the *systolic* arterial pressure, the controller uses the *mean* arterial pressure because it is less influenced by measurement problems, such as blood clots or air bubbles in the catheter, the measurement site, and spontaneous fluctuations.

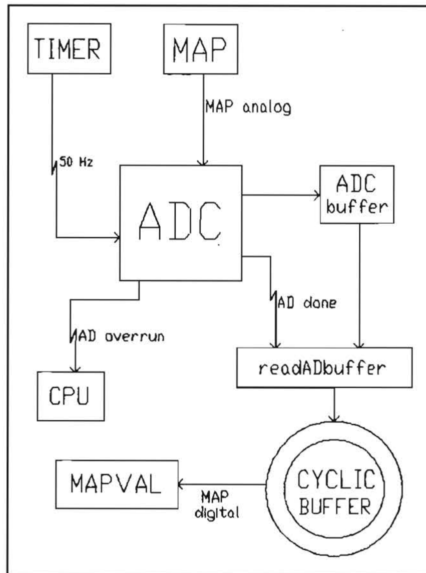


Figure 9.6. Analog to digital conversion processing.

During perfusion, the period in which the circulation of the blood is realized by a heart-lung machine, the pressure signal is almost flat. No cycle period can be detected, but an artificial cycle period of one second is imposed. During perfusion, a different validation algorithm therefore checks only 1) the signal's mean value (the MAP), and 2) the difference between the signal's minimum and maximum over a cycle.

9.3.2. The adaptive PID-controller

The controller module, the function of which is described in section 9.2.4, is activated once every 5 seconds; it receives as its input the validated 5-second-mean arterial pressure, and generates as its output a new infusion flow rate, which is then communicated to the computer controlled infusion pump by buffering a 'new flow rate' message and activating the pump's interrupt routine.

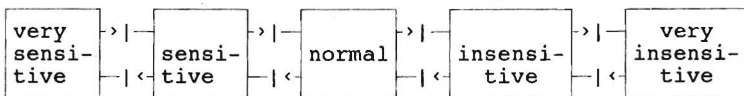


Figure 9.7. The patient's sensitivity sub-protocol.

The knowledge about the patient's sensitivity resp. the control gain is maintained in a sub-protocol (figure 9.7). The nominal sensitivity is assumed to be - 0.2 mmHg/(mg/hr).

Initially, to ensure safety, the patient is assumed to be very sensitive. The patient's sensitivity is *decreased* or *increased* according to rules that monitor the offset according to the knowledge set forth in section 9.2.4. A sensitivity increasing rule thus looks like:

```
Gain_Up_Region_0: 'Gain must be increased while in region 0'
Region_0 › 260 and Gain_Const › 180 and Setp_Const › 120
Then True: Gain_Up_Requested
```

This rule also shows some gain adaptation constraints. Not only must the pressure have been in region 0 for at least 260 seconds (the MAP always leaves this region within 260 seconds if the control gain is high enough), but the gain must also have been constant for at least 180 seconds (it takes approximately this time before the previous gain up procedure has fully taken effect) and the setpoint must have been unchanged during the last 120 seconds. The last constraint is needed in order not to confuse the region-based adaptation mechanism. Suppose that the user repeatedly changes the setpoint in such a way that the offset remains constant. This action forces a varying MAP to stay in one region for a long time, whereas a long stay in a region is meant to be interpreted as a long period of (almost) constant MAP at a certain distance from a constant setpoint.

Moreover, this rule *requests* a gain change only; some conditions (non-response situation, the gain is already at its maximum value, a transient is going on) prevent the request from being honored.

9.3.3. The supervisor module

The supervisor module has several tasks in controlling the overall behavior of the control system.

It controls the controller's mode: manual or automatic. The controller's mode is maintained in a sub-protocol, as shown in figure 9.8.

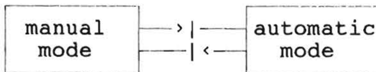


Figure 9.8. The manual vs. automatic sub-protocol.

Initially, the system is in manual mode. A switch from manual to automatic mode or back is performed when the appropriate key-press is detected.

The supervisor module must also warn if the controller cannot cope anymore, e.g. because the patient is resistant or the maximum cumulative dose has been reached. The implemented knowledge has been detailed in section 9.2.5.

It must warn the user if a pump error is detected. In case of a pump error, the system gives an auditory signal, and starts interrogating the pump at 5 second intervals in order to

detect if the problem has been solved. As soon as the pump error has been resolved, the SNP flow rate is restored to its pre-error value, and control can be resumed.

It warns when unexpected pressure transients are detected. During a positive transient, the SNP flow is kept constant at the pre-transient value. The positive transient ends when the MAP is near the setpoint again or after a certain time has passed; in the latter case the positive excursion was not a transient, and control action must be initiated to eliminate the too high MAP. During a negative transient the SNP flow is shut off. The SNP flow returns to its pre-transient value when the pressure is near the setpoint again. The zero flow period often causes a pressure overshoot. While such an overshoot occurs, the SNP flow is kept at its pre-transient value as well. The occurrence of transients is maintained in the sub-protocol of figure 9.9.

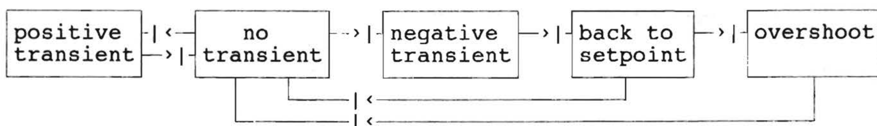


Figure 9.9. The transients sub-protocol.

Initially, the state is 'no transient'. A positive transient is detected if the MAP rises more than 20 mmHg in less than 15 seconds, or more than 25 mmHg in 30 seconds, or more than 33 mmHg in 60 seconds or more than 40 mmHg in 150 seconds; a trigger rule becomes true, the pre-transient pressure is stored, and state 'positive transient' becomes active. During a positive transient, the flow is frozen at 95% of its pre-transient value if the system is in automatic mode. If the system is in manual mode, the flow is not changed and can be adjusted manually as normal. Switching to automatic during a positive transient will freeze the flow at the level it had in manual mode. The positive transient is considered to be over as soon as the MAP is near the setpoint again or after 5 minutes have passed, whichever comes first; in the latter case the positive excursion was not a transient but lasted longer.

A negative transient is detected if the MAP falls more than 23 mmHg in less then 15 seconds, or more than 30 mmHg in 30 seconds, or more than 33 mmHg in 60 seconds. This downslope marks the start of a transient; the flow is shut off, and a reference value for the MAP, halfway between the current MAP and the pre-transient MAP, is stored. State 'negative transient' becomes active. The transition to state 'back to setpoint' is made when the MAP rises above the reference MAP; the transient is then almost ended. The flow is restored, but frozen at its pre-transient value. Now an overshoot is expected due to the zero flow period, but this overshoot does not always appear; state 'overshoot' handles the overshoot. When the MAP approaches the pre-transient MAP value again, the overshoot has ended and normal control is resumed.

In many cases the complete sequence will not happen, for instance if no overshoot occurs. In such cases, time-outs occur, which are not shown in the sub-protocol of figure 9.9. If state 'back to setpoint' or 'overshoot' is active for more than 90 seconds, no overshoot is detected

and state 'no transient' becomes active again. Similarly, if the MAP does not rise in state 'negative transient', state 'no transient' becomes active again after a 4 minute delay.

The transients sub-protocol is thus a (partial) description of *what can happen* in the signal (*states* the signal can be in), *what must be detected* to track the signal's progress through the transients (*triggers* to be evaluated), and *which actions must be performed* meanwhile.

9.3.4. The user interface

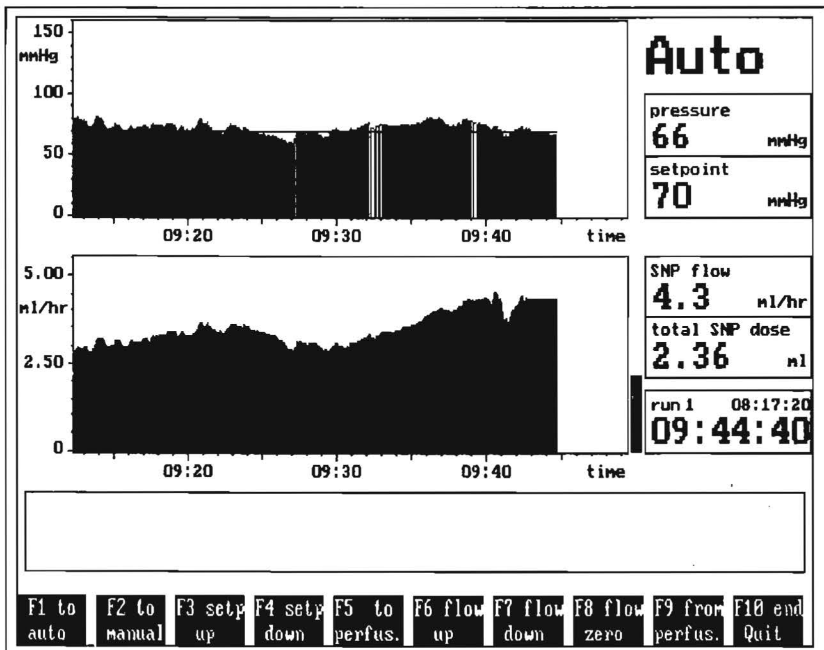


Figure 9.10. The display screen layout. This figure shows all function keys, whereas in reality only those keys are shown that are functional in the prevailing context.

To a large degree the functionality of a device or system is determined by its user interface [Coolen, 1985; Mitchell, 1987]. Since the new blood pressure controller system is both a research vehicle designed to test a knowledge base and a clinical instrument designed to stabilize the blood pressure, there are also two different classes of users: knowledge engineers and clinicians. These users have very different requirements. But if the system is to be clinically acceptable, its user interface is to be designed for clinicians; the knowledge engineering features should not interfere.

The user interface module accepts commands from the user, and it displays the controller's performance. Displayed are: the MAP and its setpoint, and the SNP flow rate,

both as trend curves over the last 30 minutes and as numbers giving the current value; warning and error messages; and the functions of the currently active keys.

The layout of the screen is globally shown in figure 9.10. The lower boxes of the screen correspond with function keys. In this figure all function keys are shown, but in practice only keys which are functional in the context of that moment are shown. The displays are almost the same in manual and automatic mode; the major difference is that the user can manipulate the flow rate only in manual mode (as a consequence, the 'flow up ' and 'flow down' keys are inactive and thus not presented on the screen in automatic mode). The blood pressure and flow rate trend displays have the same time scale. The current values are shown numerically at the right hand side of the plots. Messages and warnings can be displayed in the lower screen window; they carry a time stamp.

In manual mode, both setpoint and flow rate, in automatic mode only the setpoint can be manipulated 'up' or 'down'; in automatic mode the flow rate is computed. The 'flow zero' key switches the infusion flow rate to zero immediately (within 5 seconds). The 'to auto' and the 'to manual' keys switch from manual to automatic mode and back. Key F5 selects the 'perfusion validation', key F9 the normal (pulsatile pressure) validation algorithm. Key 10 halts the system but, to prevent errors, can do this only if in manual mode with a zero flow rate.

The keyboard has its own interrupt routine; this is set up similarly to the AD-converter interrupt routine. This was necessary to keep the displayed setpoint and flow rate up to date, so that immediate visual feedback is possible. The interrupt routine also buffers the keys and/or the number of key-presses for use by the expert system; the expert system consults these buffers and takes action within at most 5 seconds.

9.3.5. Knowledge engineering properties of the system

The knowledge engineering properties of the control system must allow a complete reconstruction of the system's internal (inferencing) and external (control) performance during the surgical procedure. This necessitates storage of several types of information.

- *Samples.* The arterial pressure signal is sampled at 50 Hz, and the total system's operation finally depends on these samples. Preliminary tests have provided much knowledge about the signal but exceptional signals may still occur, whose detailed analysis is required. The 50 Hz samples can thus be written to disk, but since the disk's capacity is limited this storage must be selective. We have chosen to use two of the keyboard's keys to initiate and stop the storage to disk of raw samples.

- *Features.* If samples are stored, the performance of the feature extraction and validation algorithm can be reconstructed because after every pulse period all feature values and their statuses are stored to disk. If no valid period is found, a time-out mechanism still generates an artificial 'period'; some of the features of such an artificial period offer

insight into the nature of the signal, in particular the signal's maximum and minimum values.

- *MAP values*. The input to the expert system is a sequence of 5 second averaged MAP values and their statuses. These are stored to disk. These data allow a complete reconstruction of the expert system's operation.

- *Conclusions*. The 'debug' output of the expert system, consisting of the conclusions after every run, has been described in section 5.6.4. This output makes it easy to analyze any particular inferencing detail using the Debugger/Tracer tool.

In addition, a knowledgeable student who attended the tests and supervised the equipment made notes of all surgical and anesthesiological maneuvers that influenced or could influence the blood pressure.

9.4. Clinical tests

The control system was tested during cardiac surgery procedures, mainly bypass cases, after approval by the Medical Ethics Committee and with informed consent of the patients. Clinical tests proceded in three phases. During the first phase all signal processing and display functions were tested, including the data validation. During the second phase (33 cases) the system was tested under 'open loop' conditions; although control was still manual, an SNP flow rate was computed by the expert system. The first and second phases also provided extra information on the ways in which clinicians controlled the SNP flow rate. The second phase was mainly used to debug the system, to correct the knowledge base where necessary, to gain sufficient confidence in the system's performance, and to familiarize the clinicians with the system. The third phase (30 cases), closed loop control, was started when we thought the system could be trusted under all conditions; during this phase the knowledge base remained unchanged.

Bypass surgery has three stages. During the first stage access to the heart is made, and blood vessels are resected from a leg. During the second stage, these vessels are used to replace defective coronary arteries. The heart is inoperative and blood circulation and oxygenation is provided by a heart-lung machine (perfusion); the resulting blood pressure signal is almost flat. The MAP is mainly controlled by the heart-lung machine and is at an exceptionally low level (around 40 mmHg). During the third stage the heart is reactivated, the chest is closed and the MAP is allowed to rise again to its pre-perfusion level.

The clinical procedure followed during the closed loop control tests was as follows:

- Prepare the infusion fluid. During the clinical tests, standard procedures were imitated as much as possible, so that at least manual control would be familiar. One consequence was to use an SNP concentration of 100 mg/l, 10 times lower than normal. This was due

to a limitation of the IMED 929 pump that was used in the tests; it can only provide a flow in increments of 1 ml/hr, whereas the pump that was normally used could deliver the flow in increments of 0.1 ml/hr. The extra dilution made the differences between the pump settings invisible to the clinicians.

- Prepare the infusion pump, flush the line and connect it to the fluid infusion line.
- Perform the calibration procedure by offering a zero pressure to the pressure transducer, start the calibration algorithm and press the calibration button on the pressure monitor. The control system is now in manual mode.
- As soon as a reliable arterial blood pressure is available, the system's automatic mode can be entered after specifying a setpoint.
- While in automatic mode, the clinician must keep the system's setpoint up to date, i.e. in agreement with the desired MAP pressure.
- While testing in automatic mode, the clinician must supervise the control system's performance; he has the options to 1) immediately switch the flow to zero and 2) to return to manual control.
- When perfusion starts, the perfusion validation algoritm must be activated; when perfusion ends, the normal (pulsatile) validation algoritm must be reactivated.
- At the end of the procedure the system must be halted.

9.4.1. Data acquisition and validation performance

The validation algorithm (see section 8.2) was tested separately. It was shown to perform adequately: all significant artifacts were detected. A small percentage of acceptable periods was flagged as invalid, but not enough to significantly influence control actions. This is due to both the 5 second averaging algorithm which delivers a valid MAP as long as at least one period within the 5 second interval is valid, and to the fact that missing upto twelve successive 5 second average values has an negligible impact on the controller's performance (see section 9.3.2). For details see Zwart [1990].

9.4.2. The control performance

The controller was generally switched to automatic soon after the arterial pressure measurement became available, and remained in control before, during and after perfusion. Occasionally the clinician considered the controller's response to be too slow; manual control was then selected, the correct flow rate was set, and automatic mode was returned to.

A major aspect of the controller's performance can be assessed by measuring the offset, the difference between MAP and setpoint, over time. Table 9.3 gives such an assessment for manual control during a total of 33 cases (85889 5 second MAP averages), and table 9.4 for automatic control during 30 cases (60970 5 second MAP averages; manual control episodes are not included) [Zwart, 1990]. The offsets are divided into 5 mmHg ranges, and the total offset counts within each range are given. In the interpretation of these results, several

factors are important to consider. In the first place is it, both for the clinician and for the controller, impossible to control the pressure if the setpoint is above the MAP (no negative flows can be given), or when the imposed maximum value limits the flow rate. In the second place, during manual control clinicians were sometimes too preoccupied with other matters to promptly mention the new setpoint that they considered appropriate in a new situation. Third, it is not always the control system's highest priority to keep the MAP close to the setpoint, e.g. when transients occur; the same is undoubtedly true for the clinician. Fourth, manual control episodes during an automatic control regime were disregarded. On average, one such episode occurred during a case; generally it lasted less than 20 seconds but its influence on the pressure was of course over a longer period. Fifth, the numbers of cases compared are too small to allow definitive pronouncements.

MAP - setpt		count	85889 MAP values
<	-25	6065	******
-25	-20	3505	****
-20	-15	5975	******
-15	-10	7429	*******
-10	-5	11403	***********
-5	0	13834	***************
0	5	11662	************
5	10	10126	**********
10	15	5503	******
15	20	4034	****
20	25	2459	**
>	25	3903	****

Table 9.3. Assessment of manual control during a total of 33 cases; number of 5 second offsets within 5 mmHg ranges.

MAP - setpt		count	60970 MAP values
<	-25	2150	***
-25	-20	2138	***
-20	-15	3703	******
-15	-10	6052	*********
-10	-5	9531	*************
-5	0	10606	***************
0	5	9494	*************
5	10	7434	**********
10	15	4608	******
15	20	2472	***
20	25	1386	**
>	25	1396	**

Table 9.4. Assessment of automatic control during a total of 30 cases; number of 5 second offsets within 5 mmHg ranges.

Another way to present these results is by noting how often the offset stayed within a zero-centered band (table 9.5). We notice from table 9.5 that automatic control is consistently better.

offset band	manual	auto
± 5 mmHg	30%	33%
± 10 mmHg	55%	61%
± 15 mmHg	70%	78%
± 20 mmHg	81%	88%

Table 9.5. Percentages of the offset in zero-centered bands in manual control and automatic control.

Averages and standard deviations of the distributions that were shown in the histograms of tables 9.3 and 9.4 are presented in table 9.6. On average, both clinicians and controller keep the MAP close to the setpoint. The controller's standard deviation is smaller, however, indicating that in automatic control the MAP is less often far from the setpoint.

	manual	automatic	
average offset	- 0.8	- 1.1	mmHg
standard deviation	18.1	12.6	mmHg

Table 9.6. Evaluation of control regimes during manual and automatic control.

The results suggest that automatic control is slightly better than manual control in keeping the MAP close to the setpoint.

9.4.3. The performance of the complete system

The implemented knowledge proved to be correct. The controller proved to be safe and could handle most cases well, except when its maximum flow rate needed to have been exceeded. Transients, both positive and negative, were recognized and processed correctly (on average, two per case). The gain adaptation worked correctly as well but was overly cautious in some cases: when the flow rate had been (almost) zero for some time, the gain was decreased, assuming a possible increase of the patient's sensitivity. This is probably unnecessary in most, maybe in all, cases.

Switching back to manual control with an audible signal after a one minute signal loss worked correctly. Usually the clinician just activated automatic mode again when the signal was restored.

The controller was almost never confronted with a steady state situation. So many other factors influence the blood pressure during cardiac surgery that the controller is only one more factor, and a slow one compared to e.g. the heart-lung machine's flow setting. Yet, it was an unproblematic one, accepted and trusted by the clinicians.

The system's user interface was appreciated for its simplicity of control and for the extra information that it provided.

The only criticism by the clinicians was that in some cases the system reacted too slowly. The clinicians considered its safe and cautious behavior, a design criterium, to be more approriate in an intensive care environment than in cardiac surgery. Future research is needed to adapt the knowledge base to the conditions that apply during this type of surgery.

9.4.4. Some rule base statistics

act	0
ask	0
test	46
eval	88
memo	3
state	41
total	178

Table 9.5. Number of rules of each type in the controller's knowledge base.

The blood pressure controller knowledge base is large. It is a SIMPLEXYS language text file of 1499 lines (excluding Pascal code in libraries) containing 178 rules (for only 4 of which an initial value is specified), 100 ON statements, and 516 lines of Pascal code for the INIT and EXIT sections. The number of rules of each type is given in table 9.5. There are also 2251 lines of Pascal code in several libraries. The number of Pascal code lines in the Inference Engine and each of the libraries is given in table 9.6.

Inference Engine	1382
global variables	58
control parameter initializations	152
keyboard input routines	61
text mode display routines	65
graphics mode display routines	689
blood pressure validation routines	526
analog to digital converter routines	258
infusion pump communication routines	442
total	3633

Table 9.6. Number of lines of Pascal code of Inference Engine and libraries.

The output of the SIMPLEXYS Rule Compiler is a set of files containing Pascal code. The length, in number of Pascal lines, of each of these files is given in table 9.7. Table 9.8 gives the times that the Rule Compiler and its extensions need to translate and check the knowledge base on a PC-AT type computer.

run time debug options	2
specification of libraries	2
DECLS, INIT and EXIT sections	516
inference engine tables (arrays)	685
code of TEST rules	54
code of DO's	216
code of history comparisons	123
total	1598

Table 9.7. Number of lines of Pascal code of the Rule Compiler's output.

Rule Compiler	47 sec
Semantics Checker	3 min 16 sec
Protocol Checker	3 min 6 sec
Compilation time	7 min 9 sec

Table 9.8. Timing of the Rule Compiler and its extensions.

The internal (Pascal) format of the knowledge base plus Inference Engine together therefore form a Pascal program with a total length of 5231 lines of Pascal code. The Pascal compiler compiles this to an MS-DOS executable file with a size of 111520 bytes, and it estimates a run time memory requirement of 122624 bytes (81344 bytes code, 41280 bytes data).

Initially, the expert system's average execution time for a run was approximately 5 seconds, of which about 4 seconds was taken in scrolling the MAP and SNP trend displays. Because the time allowed for a run was only 5 seconds, this long average processing time threatened to exhaust the processor's capacity, and might even exceed it in worst case situations. To avoid this, and also to limit the percentage of the time during which scrolling takes place, we decided to scroll the MAP and SNP displays not every sample but once every 5 minutes, leading to an estimated average 20% use of the processor's (8 MHz PC-AT) capacity.

9.4.5. SIMPLEXYS as a knowledge engineering tool

In this section the appropriateness and convenience of SIMPLEXYS as a knowledge engineering tool in the design of the blood pressure controller are discussed. Most findings are based on interviews and discussions with the system's knowledge engineer, Lammers. Quoted texts are his.

Language aspects. Rules are self-documenting; as a knowledge chunk, it is 'a good compromise', neither too large nor too small. The same is true for the rule base as a whole. 'Looking at the rule base, after working on other projects for some time, is sufficient to be immediately aware again of the implemented knowledge.' However, 'enumerated types would be nice' because in several cases they could simplify the knowledge base.

ON statements appear to have a too simple syntax. In many cases, a more complex ON statement syntax would simplify the protocol part of the knowledge base. The major problem is that the FROM list currently cannot specify an arbitrary expression (of STATE rules). A syntax like

```
ON X FROM A or B or C or D TO E
```

where A, B, C, D and E are STATE rule lists, would allow one ON statement to replace four. Another observation is that frequently, when a trigger fires, one or more other tokens become irrelevant and should disappear. A syntax like

```
ON X FROM A TO B; RESET C
```

where A, B and C are lists, would again decrease the number of ON statements significantly. Neither ON statement syntax change would have semantic consequences except better readability of the protocol.

'It is fortunate that rule histories can be freely used in expressions.' But in some cases, manipulating the rule history in more complex ways might be helpful. Resetting the history counter without changing the rule's value could sometimes help in answering questions like 'how long ago did X happen *last*?' where currently an extra dummy STATE rule is necessary to achieve this. It is also impossible to directly inquire how long a certain conclusion has been *false*. Another problem is that a history counter is unexpectedly reset when its rule is not evaluated. It is also difficult to program something like 'if rule P has been true for Q seconds, then do R every S seconds', except at a low level, where it is clumsy and unsafe.

Rapid prototyping aspects. Initially, design was bottom-up. 'I started with a bare rule base. Most of its Pascal code was available, as well as a set of graphics, filing, keyboard and simulation routines.' These had been separately developed and tested by others. 'It was surprising how fast a first prototype was built, and at least as surprising that it worked immediately. Converting an idea or concept into SIMPLEXYS rules is easy and error-free. It is also easy to add knowledge chunks and to temporarily disable some chunks to test others. The tracer/debugger is not needed to do this. In the early phases of the project, rule base changes were always additions of missing or incomplete knowledge, *not* repair of erroneous knowledge. After the first clinical tests, some partial repair became necessary.' This repair was easy, however, because the *structure* of the knowledge base did not change much.

Modularity aspects. The program as a whole is constructed in a number of ever more 'symbolic' layers:

- interrupt routines for the AD conversion, keyboard handling and pump communication;
- the validation routine that delivers the MAP;
- simple MAP-based calculations such as MAP filtering, MAP slope determination and calculation of the SNP flow rate;
- basic rules that test (possibly filtered) data, parameters, flags and key presses;
- evaluation rules that combine basic rules and/or other evaluation rules;
- the protocol.

Higher layers always refer to lower ones, except the highest two, which refer to each other. STATE rules and their goals and trigger rules represent the system at its highest level of abstraction, and it is often unclear (and unimportant) at this level of abstraction which lower layer rules will need to be evaluated.

In later phases the design had more of a top-down nature in which changes mostly took place in the two highest layers, with an occasional excursion to lower ones.

Tracing/debugging aspects. 'For complex problems the tracer/debugger is indispensable. Not so much to discover what happened and why, but to see what did *not* happen, or rather, what exactly was the cause that a certain decision was not arrived at.' It was often important to discover which rules were and which were not evaluated in a certain context. The tracer/debugger provides this type of information.

9.5. Conclusions

SIMPLEXYS was an appropriate tool in the design of the blood pressure controller, both in the design of the knowledge base and in its testing, debugging and maintenance. The complete freedom to design interfaces with the outside world, including those that use interrupt routines, is a necessity. Knowledge acquisition remains the bottleneck in the design of expert systems. Yet, SIMPLEXYS offered support rather than force the knowledge base designer into inappropriate schemes. The protocol, in particular, was a valuable construct.

The automatic blood pressure controller functioned correctly in all respects, according to its design criteria. It is impossible to decide whether we encountered 'difficult' cases in the closed loop control phase of this study; except for the somewhat overly cautious behavior of the controller (in a cardiac surgery context), no uncontrollable situations were discovered.

The controller uses a model which is partly black box, i.e. based on a mathematical description of the input-output relation of the system, and partly mechanistic, i.e. based on knowledge of the internals of the system. Such intermediate models lead to unique controllers which are often not optimal in a theoretical sense (they do not try to optimize some performance criterium), but robust (they are adequate under wide operating conditions despite an imprecise model). As a consequence, their performance is often difficult to

measure. The difference between MAP and setpoint is one measure, but not the only one, since the pressure need not be kept at a fixed level but can be allowed to fluctuate within reason.

There are other criteria. One constraint is that, for reasons of safety, incremental increases in infusion rate are limited. Also, overdosage has to be avoided at all costs. The control system appears to achieve this better than the clinicians, indicating that the advantage of computer control may not necessarily be in improved control but rather in the support it provides.

The system is simple to interact with, unproblematic in use, and in special cases manual control can take over without much effort.

10. Conclusions

In this chapter we first survey the solutions that SIMPLEXYS offers in the domain of real time expert systems. We then appraise its success as a new computer language, including the software that surrounds the language. We then return briefly to what SIMPLEXYS meant in the design of the blood pressure controller of chapter 9. A possible simplification of SIMPLEXYS and a potential new application domain will be described next. We finish with an overall conclusion.

10.1. SIMPLEXYS as a real time expert system

In section 2.2.3, some of the current problems in real time expert systems research were reviewed. These were:

- Traditional tools are not well suited to real time applications. Many inferencing algorithms are exponential time. An immediate goal should be the development of high-performance inference engines that can guarantee response times.
- Current shells are as suitable in the real time domain as Prolog for a number-crunching application or Fortran for a symbolic processing application.
- 'Real time expert systems are real hard to develop'.

SIMPLEXYS offers some solutions, notably:

- The Inference Engine has a linear time algorithm, resulting in a high execution speed; this is due to the elimination of searching during the inferencing process. Many expert systems are currently *production rule* based. Knowledge engineers who have experience with production systems will, because of the resemblance, probably have little trouble becoming familiarized with the SIMPLEXYS format. They can immediately profit from the high performance of SIMPLEXYS, because it has been shown (in section 4.11) how a production rule based expert system can be converted into the SIMPLEXYS format.
- The system's worst case response time can be estimated rather accurately; moreover, the response time will generally not vary much from run to run, allowing a uniform processor utilization, so that the system's performance does not become brittle on modifications of the problem specification and type or quantity of data; in particular, no tuning will be necessary, nor mechanisms to focus attention; since garbage collection does not exist in SIMPLEXYS expert systems, it is not a problem; operation is continuous and predictable.
- SIMPLEXYS offers optimal environment utilization: compiled instead of interpreted code.
- A general problem solving framework, so that no ad hoc methods need to be reinvented for each new problem.
- Although thus far only two major applications have been designed with the SIMPLEXYS toolbox, a first impression is that SIMPLEXYS seems to offer an adequate capacity for

handling time sequencing information in the type of monitoring applications it is intended for.
- SIMPLEXYS expert systems integrate efficiently with conventional software; they efficiently integrate numeric with symbolic computing.
- SIMPLEXYS applications can integrate with a real time clock.
- Although there are no facilities for handling asynchronous inputs nor a way of handling software-hardware interrupts (see section 3.4 on why interrupts have no place in SIMPLEXYS), conventional software that performs such functions is easily interfaced with (see section 9.3.1 on an example on how to interface with interrupts); in particular, SIMPLEXYS expert systems can efficiently obtain input from external non-human stimuli.
- Although the SIMPLEXYS inference engine has thus far not been formally validated, its trouble-free prolonged use gives adequate assurance of its reliability.
- SIMPLEXYS offers a variety of methods to verify and validate knowledge bases.
- SIMPLEXYS can run on PC-like hardware that is built for harsh environments.

Thus SIMPLEXYS offers all the tools that are necessary to design, build and test patient monitoring oriented and other real time expert systems. It exists both in a Pascal and a C version. Both are fully functional, as was demonstrated by the expert systems built with the SIMPLEXYS tools: the blood pressure controller of chapter 9 uses the Pascal version, and the intelligent alarms system [van der Aa, 1990] uses the C version.

SIMPLEXYS thus solves some of the current problems in real time expert systems research:

- The SIMPLEXYS tools are well suited to real time applications. SIMPLEXYS has a high-performance inference engine, and worst case response times can be estimated well.
- The SIMPLEXYS toolbox is suitable in the real time domain, both for number-crunching applications and for symbolic processing applications.
- SIMPLEXYS real time expert systems are *not* hard to develop.

10.2. SIMPLEXYS as a programming language

In section 3.4 we noted that the primary goal of any programming language is to allow the programmer to formulate his thoughts in terms of abstractions suitable to his problem, to build an 'abstract machine'. Two applications have demonstrated that SIMPLEXYS has the right abstractions to encode knowledge within the domain of monitoring applications. The *protocol* describes the dynamic aspects of the knowledge base at a high level of abstraction; it also specifies the (often context-dependent) *goals*, the abstractions of the tasks to be performed. The *rules* have demonstrated to be appropriate constructs to formally describe chunks of knowledge. And the structure of SIMPLEXYS *programs*, finally, has been shown to correspond well with the tasks that a real time expert system has to execute.

The second goal is that the 'abstract machine' must actually run and solve the problems it was designed to solve. The issue is now efficiency, both in terms of code size and speed of execution. The abstractions must be efficient. In SIMPLEXYS, they are.

A third goal is that the 'abstract machine' must be so logically built, that a compiler can check the legality of the program statements. In SIMPLEXYS, not only single program statements are checked, but also the program as a whole. Because deep semantic knowledge is necessary to accomplish the latter, this type of testing cannot be complete. Yet, in SIMPLEXYS it can be much more thorough than usual, because of the simplicity and formal nature of the constructs. Because the links between these constructs are explicitly known, the compiler can discovers a variety of *global* errors in the knowledge base. This is possible because *a priori* knowledge exists about some of the *semantic* properties of a knowledge base. This makes it possible for the compiler, for example, to detect that the end of a protocol cannot be reached because of a deadlock situation or that a goal cannot be evaluated because its definition is self-referential. This type of checking makes it possible to discover a variety of errors in the knowledge base in its early design stages without extensive testing.

A fourth goal is to have a good compiler. The SIMPLEXYS compiler is reliable; it performs a syntax check against every rule of the language, it translates correctly and takes care that no incorrect program crashes the compiler. Also, although higher level static and dynamic correctness checks can be slow, the SIMPLEXYS Rule Compiler itself compiles at a reasonable speed, it generates efficient code and the execution cost of the code is reasonably predictable. The compiler is compact. It also provides a simple and effective interface to the environment to programmers who are familiar with Pascal or C.

10.3. The SIMPLEXYS toolbox

The goals, set forth in section 4.1, have been realized by the currently existing tools.

- SIMPLEXYS applications are small enough to run on a PC and yet be fast enough; the storage of the knowledge is very compact and the inferencing method is ideally suited to real time work.
- Applications can have custom-designed user and device interfaces; the easy interface with Pascal allows anything, including efficient computations.
- A good estimate of the worst case performance can be obtained. Due to the linear time algorithm, moreover, the ratio between worst case run time and average run time is relatively small.
- SIMPLEXYS expert systems are easy to program and use.
- The available knowledge structures are well geared to the type of applications.
- The knowledge base can be checked thoroughly in a number of ways, so that a high level of confidence in its correctness can be carried from the very early design stages to the final product.

10.4. The blood pressure controller

SIMPLEXYS was a proper tool in all phases of the construction of the sodium nitroprusside blood pressure controller: in the design of the knowledge base, in its testing and debugging and in its maintenance phase. The freedom to design interfaces with the outside world, including those that use interrupt routines, is a necessity in a patient monitoring context. SIMPLEXYS, because it is especially suited to this type of application, offered support rather than force the knowledge base designer into inappropriate schemes. The protocol, in particular, proved to be a valuable construct.

The automatic blood pressure controller functioned correctly in all respects, according to its design criteria. Its robustness is convincing: it is impossible to decide whether we encountered 'difficult' cases in this study. Due to the cautious behavior of the controller (maybe somewhat overly cautious in a cardiac surgery context), no uncontrollable situations were discovered. The system is simple to interact with, unproblematic in its use, and it offers valuable extra information to the user. In special cases it is easy to temporarily take over with manual control.

10.5. SIMPLEXYS simplified

A review of the knowledge bases of both the blood pressure controller described in chapter 9 and the intelligent alarms system [van der Aa, 1990] brought to light that the truth value PO (unknown) was not exploited; the acknowledgment that knowledge was missing was, if necessary, handled by extra rules. This can have several reasons. One is that the knowledge engineers are not used to its availability; they are not 'experts' in three-valued logic and hence understandably reluctant to use unfamiliar building blocks in what must become high quality products. If this is so, we might expect that sooner or later they will 'discover' the possibilities of three-valued logic and start to use its capabilities.

Another reason might be that any other logic than the standard true-false type is really too difficult for knowledge engineers to comprehend, because they cannot fully fathom its global repercussions. In section 4.6 we discussed why we did not want to introduce a more complex logic than the one actually implemented. Maybe those very same arguments can show that the truth value PO is dispensable as well.

With only two major applications built, it is as yet impossible to decide whether to maintain the truth value PO in future versions of SIMPLEXYS or not.

Suppose, however, that the truth value PO is abolished from the SIMPLEXYS language. This has vast repercussions throughout the language and its implementation. Evaluation of a rule then always leads to a conclusion of either TR or FA, as in Boolean logic. The operators MUST, POSS and ALT become superfluous. The compiler thus becomes simpler, in particular the completion of expressions described in section 6.5.2.3.3. Checking becomes simpler, because elimination of the ALT operator means that no alternative evaluation paths need to be traversed any more. And inferencing becomes slightly faster as well. Such a

simplified (and faster, but less expressive) SIMPLEXYS might be adequate for many applications.

10.6. SIMPLEXYS in hardware

This simplified SIMPLEXYS might also become a tool for the design of very high speed *hardware* expert systems or 'programmable logic controllers' [PLCs; Vingerhoeds et al, 1989], the design of which can be carried through using all the facilities of the SIMPLEXYS toolbox, but whose final delivery takes the form of a custom hardware unit. The static AND-OR net, which results from the simplification, is easy to implement using standard logic gates. And the dynamic net, the protocol, shows a remarkable similarity with what Hill and Peterson [1978] call an AHPL organization, which is implemented using clocked flip-flops.

Interfacing with the outside world, which is currently done through Pascal code, then becomes the bottleneck where speed is concerned. In simple cases, input and output can be realized by (analog and digital) electronics; a TEST could be performed by a comparator, executing a DO could be as simple as setting a flip-flop. In more complex cases, the Pascal (or any other familiar programming language) code cannot be dispensed with, and would require for its execution an extra high speed uni- or multi-processor system, depending upon the necessary level of performance.

10.7. General conclusions

In both major applications of SIMPLEXYS, the blood pressure controller described in chapter 9 and the intelligent alarms system [van der Aa, 1990], knowledge acquisition was by far the most difficult and time-consuming activity. One can only hope, that the knowledge representation language and the mechanism that translates the knowledge base into a delivery system do not present additional difficulties but instead offer assistance. The knowledge engineers, as well as the results obtained with the finished products, confirmed that SIMPLEXYS is a convenient toolbox. Its language offers a convenient framework (the *protocol* and the *goals*) to express the problem at a high level, the necessary elements to conveniently chunk the knowledge (the different *rule types*), and the required constructs to efficiently interface with the outside world. Its compiler delivers code which can be executed at high speed, and it offers a variety of tests to check for the integrity and correctness of the knowledge base. Several types of errors that cannot be found at compile time can be discovered by the Inference Engine at run time. And if the expert system still does not perform as expected, the tracer/debugger may help to discover 'deep' semantic errors.

The overall conclusion is that from an engineering point of view, the SIMPLEXYS experiment is a success if tested according to the criterium that it supports the construction of fast, inexpensive, compact, robust and safe real time expert systems. From an AI-theoretical point of view, the SIMPLEXYS experiment is a success in that it provides a comprehensive framework to express and solve a class of (process supervision and control) problems.

Appendix 1. The SIMPLEXYS syntax

Meta-descriptors

The formal description of the (Pascal version) syntax of SIMPLEXYS uses the following meta-descriptors:

1. the symbol ::= indicates a definition;
2. an item between { and } is optional;
3. a * behind an item indicates 1 or more repetitions;
4. a * behind an optional item indicates 0 or more repetitions;
5. the symbol | indicates an alternative;
6. text between [and] is a comment only;
7. the symbol NL indicates the 'new line' symbol.

SIMPLEXYS is *line-oriented.* This has several reasons, but the most important one is, that it forces a good program layout. This makes the program easier to overview and comprehend. Also, expressions cannot become too long and complex; shorter expressions often also lead to more efficient evaluations, since they invite the re-use of sub-expressions.

The SIMPLEXYS language is case-insensitive, i.e. 'AND' and 'and' mean the same thing, as well as 'And', etc.

The SIMPLEXYS syntax

```
program         ::= {DECLS NL declarations NL}
                    {INITG NL code NL}
                    {INITR NL code NL}
                    {EXITR NL code NL}
                    {EXITG NL code NL}
                     RULES NL rules NL
                     PROCESS NL transitions NL

declarations    ::=  Pascalcode [declarations of types, variables,
                                 functions and procedures only]
code            ::=  Pascalcode [executable statements]

rules           ::=  rule NL
                  |  rules rule NL
rule            ::=  ruleheader NL
                     rulebody NL
                    {ruleinit NL}
                    {consequences NL}

ruleheader      ::=  rulename : ruletext
rulename        ::=  alphanumeric {xalphanumeric} *
alphanumeric    ::=  0 | 1 | .. | 9 | A | .. | Z | a | .. | z
xalphanumeric   ::=  alphanumeric | _
ruletext        ::=  string
```

```
string          ::=  ' {character} * ' [but not character ']
                 |   " {character} * " [but not character "]

rulebody        ::=  FACT
                 |   ASK
                 |   TEST testline
                 |   TEST NL testbody NL ENDTEST
                 |   BTEST boolean_expr
                 |   MEMO
                 |   STATE
                 |   expression

testline        ::=  Pascalcode [statement assigning to var TEST]
testbody        ::=  Pascalcode [statement(s) assigning to TEST]
boolean_expr    ::=  Pascalcode [boolean expression]
expression      ::=  term
                 |   expression op2 term
term            ::=  rulename
                 |   rulename historyop numval
                 |   op1 term
                 |   ( expression )
op1             ::=  NOT | MUST | POSS
op2             ::=  AND | UCAND | OR | UCOR | ALT
historyop       ::=  = | <> | < | <= | > | >=
numval          ::=  ( <Pascal numeric expression> )

ruleinit        ::=  INITIALLY TR | FA | PO

consequences    ::=  {consequence NL} *
consequence     ::=  THEN | ELSE | IFPO conseq
conseq          ::=  ruleconseq | DOcode | goals
ruleconseq      ::=  TR | FA | PO : rulename *
DOcode          ::=  DO Pascalcode [one executable statement]
                 |   DO NL Pascalcode [statement(s)] NL ENDDO
goals           ::=  GOAL : rulename *

transitions     ::=  transition NL
                 |   transitions transition NL
transition      ::=  ON rulename FROM statelist TO estatelist
statelist       ::=  staterule *
estatelist      ::=  statelist
                 |   * [denotes the empty list]
staterule       ::=  rulename
```

Operator priorities

In expressions, operator priorities are thus as follows:

highest priority:	history operators
medium priority:	monadic operators
lowest priority:	dyadic operators

If no confusion can arise, unnecessary parentheses in expressions may be discarded. For example, (((R1 AND R2) AND R3) AND R4) can be rewritten as R1 AND R2 AND R3 AND R4, keeping in mind that the application of dyadic operators is from left to right[1].

Semantic exceptions

For semantic reasons, the following exceptions are enforced:

1. FACT rules cannot be goals; their conclusion is assumed fixed *and known*, so there is no logical reason to evaluate them.
2. FACT rules cannot be assigned to with THELSEs; their conclusion is assumed fixed, so it should not be possible to give them (a different) conclusion.
3. MEMO rules cannot be goals; they have a conclusion already (the one inherited from the previous run), so there is no logical reason to evaluate them.
4. STATE rules cannot be goals; they have a conclusion already (the one inherited from the previous run), so there is no logical reason to evaluate them.
5. STATE rules cannot be assigned to with THELSEs; the assignment takes place through ON statements.
6. STATE rules can only have conclusions TR and FA; the context must be known.
7. An INITIALLY TR to at least one STATE rule must exist; otherwise no context is initially active.

Reserved words

The reserved words DECLS, INITG, INITR, EXITR, EXITG, RULES, PROCESS, NOT, MUST, POSS, AND, UCAND, OR, UCOR, ALT, FACT, ASK, TEST, BTEST, ENDTEST, MEMO, STATE, INITIALLY, THEN, ELSE, IFPO, TR, FA, PO, GOAL, DO and ENDDO must not be used as rule names, nor their equivalents if some or all letters are changed into lower case; the Rule Compiler converts lower case to upper case when checking for reserved words.

Comments

Comments can appear in the text in any position where a space character would be permitted. Multi-line comments are not allowed. Comments are enclosed by the characters '{' and '}'. Comments must not be nested.

---

[1] Monadic operators are applied from right to left; thus the expression MUST NOT POSS R5 is evaluated as MUST (NOT (POSS R5)).

Appendix 2. Additional operators

It is possible to 'define' many additional operators using the existing operators. In the SIMPLEXYS three-valued logic, 3^3 = 27 different monadic and 3^3^3 = 19683 different dyadic operators could theoretically be defined. Not all these operators are of course equally meaningful. Experiments brought to light that if additional operators were defined in the SIMPLEXYS syntax, due to unfamiliarity they would not be used except in very special cases. These special cases were deemed not important enough to make the language more complex.

If necessary, therefore, the construction of additional operators can be done in terms of the existing ones. An example: a construction

```
A unless B
```

might be needed, whose conclusion is conclusion A unless conclusion B can be demonstrated to be TR, in which case the conclusion must be FA. An operator *unless* is not defined in SIMPLEXYS, but it can be constructed from e.g.

```
A and not must B
```

as can be verified:

B	must B	not must B	A and not must B
FA	FA	TR	A
PO	FA	TR	A
TR	TR	FA	FA

Appendix 3. SIMPLEXYS utilities

The following symbols/utilities are predefined for the user:

type bool = (TR, FA, PO, UD);

function ASKyn (t: string): boolean; prints string t and waits for a 'y' or 'n' followed by a 'return' typed in at the keyboard.

function ASKval (t: string): bool; prints string t and waits for a 'y', 'n' or '?' followed by a 'return' typed in at the keyboard.

function ASKint (t: string; imin, imax: integer): integer; prints string t and waits for an integer followed by a 'return' typed in at the keyboard; the input must be between imin and imax (inclusive).

function ASKword (t: string; imin, imax: word): word; prints string t and waits for a non-negative integer followed by a 'return' typed in at the keyboard; the input must be between imin and imax (inclusive).

function ASKreal (t: string; imin, imax: real): real; prints string t and waits for a real number followed by a 'return' typed in at the keyboard; the input must be between imin and imax (inclusive).

procedure dump (t: string); prints string t, followed by all currently known results (rules with conclusions TR, FA or PO); useful while debugging.

procedure fatal_error (t: string); prints an error message and halts the system; can be used when a fatal error occurs.

var _error: boolean; initially, _error is given the value *false*; as soon as an inconsistency error occurs, _error is given the value *true*; the user can intercept the error (e.g. in a TEST rule) and end the expert system's operation; the Inference Engine's default action is to write an error message to file 'simplex.err' and continue.

Appendix 4. The Inference Engine's main data structures

In the Inference Engine's data structures, shown in table 1, _N is the total number of rules in the knowledge base, _F the total number of ON statements, and _E, _T, _B and _O are appropriate array dimensions.

rule storage	
_RULENAME: array [1 .. _N] of string	rule names
_RULETEXT: array [1 .. _N] of string	rule texts
_RULETYPE: array [1 .. _N] of 0..5	rule types
_IVALUE: array [1 .. _N] of bool	initial values
_R: array [1 .. _N] of bool	current values
_S: array [1 .. _N] of bool	previous values
_EVINDEX: array [1 .. _N] of 0.._E	index into _EVSTORE
_TTINDEX: array [1 .. _N] of 0.._T	index into _TTSTORE
_TBINDEX: array [1 .. _N] of 0.._T	index into _TBSTORE
rule related token storage	
_EVSTORE: array [1 .. _E] of word	expression tokens
_TTSTORE: array [1 .. _T] of word	THELSE tokens
_TBSTORE: array [1 .. _B] of word	reversed THELSE tokens
ON statement storage	
_TRIGGER: array [1 .. _F] of 1.._N	trigger rule numbers
_FRINDEX: array [1 .. _F] of 1.._O	FROM list index
_TOINDEX: array [1 .. _F] of 1.._O	TO list index
FROM/TO list storage	
_ONSTORE: array [1 .. _O] of 0.._N	FROM and TO lists

Table 1. The Inference Engine's tables.

Indices having a zero value represent a null pointer, e.g. if a rule has a zero _TTINDEX entry, it has no THELSEs. Zeroes are also use to indicate the *end* of a list, e.g. in _EVSTORE, where a zero marks the end of an expression, and in _ONSTORE, where a zero marks the end of a FROM or TO list.

The storage of a rule's information

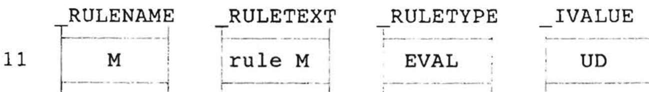
An example will demonstrate how a rule's information is stored. We consider the storage of rule M's information from the following example:

```
M : 'rule M'            B: 'rule B'
H AND (L OR P)          F AND P
THEN TR: K, F           THEN FA: M
```

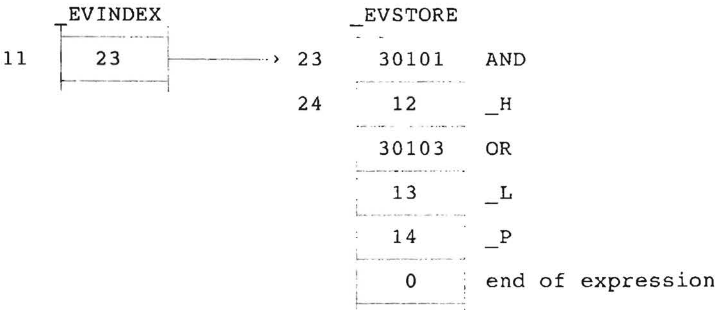
The Rule Compiler first assigns a sequence number to the rules, in the order of occurrence. The names of the rules are preserved, so that the user can access any information through Pascal code; they are preceded by an underscore, however. Assume that 10 other rules have been processed already. The sequence number assignments thus become:

```
_M := 11;   _H := 12;   _L := 13;   _P := 14;
_K := 15;   _F := 16;   _B := 17;
```

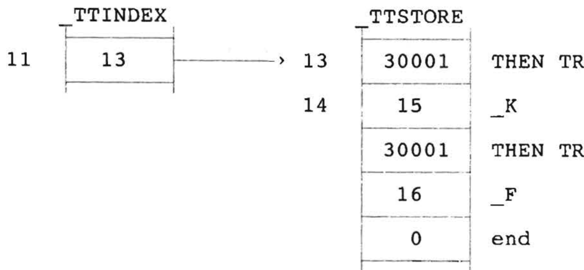
Then the rule's name, text string, type and initial value are stored. Since rule M does not have an INITIALLY, and since it is an evaluation rule, its initial value is marked UD.



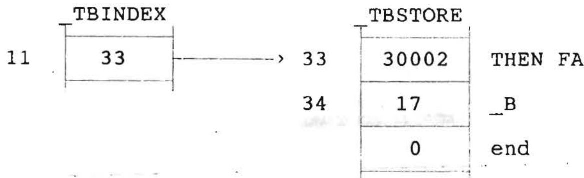
Then the Rule Compiler tokenizes and stores the rule's expression into array _EVSTORE, including a null token to indicate the expression's end; the start position of the expression is stored into _EVINDEX. Assume that the first free location in _EVSTORE is position 23. If the rule is not an evaluation rule, it has no expression, and a null is stored into _EVINDEX and nothing is stored in _EVSTORE.



Next, the Rule Compiler tokenizes and stores the rule's THELSEs into array _TTSTORE, including a null token to indicate the end; the start position is stored into _TTINDEX. Assume that the first free location in _TTSTORE is position 13. If the rule has no THELSEs, a null is stored into _TTINDEX and nothing is stored in _TTSTORE.



Finally, the Rule Compiler tokenizes and stores other rules' THELSEs to this rule into array _TBSTORE, including a null token to indicate the end; the start position is stored into _TBINDEX. Assume that the first free location in _TTSTORE is position 33, and that only rule B has a THELSE to rule M. If there are no THELSEs to the rule, a null is stored into _TBINDEX and nothing is stored in _TBSTORE.



The storage of an ON statement's information

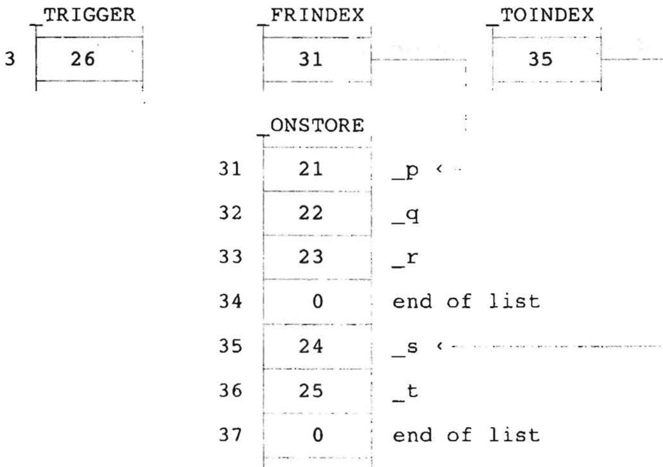
Next, we consider the storage of the following ON statement:

```
ON u FROM p q r TO s t
```

If the knowledge base is correct, all rules have already been found, their information stored, and assigned a sequence number. Assume the following sequence number assignments:

```
_p := 21;    _q := 22;      _r := 23;
_s := 24;    _t := 25;      _u := 26;
```

The trigger rule's sequence number is stored in _TRIGGER, the FROM and TO lists will both be stored in _ONSTORE, while indices to the lists will be stored in _FRINDEX and _TOINDEX. Assume that this is the third ON statement, and that the first free location in _ONSTORE occurs at position 31.



The *conclusion* of each rule is maintained in two arrays, _R and _S. Array _R stores the rule's *current* conclusion, array _S stores the conclusion that was obtained in the previous run. This organization is necessary for several bookkeeping reasons. Examples are:

- A MEMO rule's conclusion is *read* from _S and *written* to _R. The value of its _R entry varies during the run. It is tagged UD at the start of each run. A THELSE to a MEMO rule writes to _R; subsequent THELSEs to the rule check the rule's _R entry in order to detect conflicting assignments. The value of its _S entry does not, of course, change during the run.

- A rule's history counter is reset when that rule's _R entry is assigned a value different from that in the previous run, i.e. if its _R entry is different from its _S entry.

At the start of the first run, _R and _S are set up according to default values (see section 5.3.2.3), which can be modified by any initial values specified by the INITIALLYs of the knowledge base.

Appendix 5. The Inference Engine's main procedures and functions.

The Inference Engine's main procedures/functions are the following three, which are mutually recursive:

procedure _thelse (rule: 1 .. _N);
executes all the rule's THELSEs

procedure _setrule (rule: 1 .. _N, value: bool);
assigns value to the rule's conclusion without evaluation, but checking for an assignment conflict

function _evalrule (rule: 1 .. _N);
evaluates rule; returns its conclusion

Executing a THELSE is done by procedure _thelse, which has the following code:

```
value := _R [rule];
ptr := _TTINDEX [rule];
if ptr = 0 then
  exit; {exit procedure: the rule has no THELSEs}
token := _TTSTORE [ptr]; ptr := ptr + 1;
arg   := _TTSTORE [ptr]: ptr := ptr + 1;
while token <> 0 do
begin
  case token of
    thentr:     if value = TR then _setrule (arg, TR);
    thenfa:     if value = TR then _setrule (arg, FA);
    thenpo:     if value = TR then _setrule (arg, PO);
    thengoal:   if value = TR then dummy := _evalrule (arg);
    thendo:     if value = TR then _FDOS (arg);
    ... {same for elsetr etc, ifpotr etc}
    else    fatal_error ('invalid token in THELSE list') {this
            should not occur but is checked anyway}
  end case;
  token := _TTSTORE [ptr]; ptr := ptr + 1;
  arg   := _TTSTORE [ptr]: ptr := ptr + 1;
end;
```

Procedure _thelse uses procedures _FDOS and _setrule and function _evalrule; procedure _FDOS executes a DO and its structure is analogous to function _FTEST, described in section 5.6.1. Procedure _setrule does the following, depending on the type of the rule:

- If the rule type is FACT or STATE, this is a fatal error; trying to assign to a FACT or STATE rule is illegal (this should not occur because the Rule Compiler flags this as an error; for safety reasons the check is executed anyway).

- If the rule type is MEMO, then, if _R [rule] is marked UD, the new conclusion is assigned. Otherwise, a conclusion has already been assigned in the current run and the new assignment must be consistent with it. If the new assignment is different, this is flagged as an error, and the new assignment is not carried out. If the new assignment is the same, there is no problem (and of course the assignment does not need to be carried out).

- If the rule type is ASK, TEST or evaluation, then, if _R [rule] is marked UD, the new conclusion is assigned. Otherwise, a conclusion has already been assigned in the current run and the new assignment must be consistent with it. If the new assignment is different, this is flagged as an error, and the new assignment is not carried out. If the new assignment is the same, there is no problem (and of course the assignment does not need to be carried out). If the rule's conclusion becomes different from that of the previous run, the history counter is updated (_HISTORY [rule] := _time), and the rule's THELSEs are executed by calling procedure _thelse.

An important run time consistency check is thus: once a rule's conclusion is evaluated (to TR, FA or PO) it cannot be changed again to another conclusion in that same run. Trying to assign a different conclusion to that rule leads to an error message and the assignment is not executed (by default, this type of error is not fatal, but the user may want to intercept it and halt the system). This inconsistency is a result of either mutually inconsistent rules (however, inconsistency of rules can frequently be detected at compile time; see chapters 6 and 7 on semantic and protocol checks) or of inconsistent answers (to ASK rules) or data (to TEST rules).

Function _evalrule does the following:

1. If the rule's conclusion is TR, FA or PO, return that conclusion and exit. Step 1 is all that is ever needed in the retrieval of FACT, MEMO and STATE rule's conclusions; in the case of ASK, TEST and evaluation rules, step 1 is all that is needed if the rule has been evaluated before in the same run.

2. The conclusion is now marked UD; evaluate the rule: ASK a question, TEST data, evaluate an expression. The code is:

   ```
   case ruletype [rule] of
     ask:   value := _ASKval ('Is true: ' + _RULETEXT [rule] + '?');
     test:  value := _FTEST (rule);
     eval:  value := _evalexp (rule);
   end {case};
   ```

   If the result is now TR or FA, then skip step 3.

3. The conclusion is now PO; check if TR or FA conclusion can be obtained from a THELSE TR/FA from an other rule that has a conclusion marked UD; do this by traversing list _TBSTORE starting at _TBINDEX [rule], if it has entries; the main code resembles that of procedure _thelse:

```
ptr := _TBINDEX [rule]; if ptr = 0 then exit;
token := _TBSTORE [ptr]; ptr := ptr + 1;
trule := _TBSTORE [ptr]; ptr := ptr + 1;
value := PO;
while (token <> 0) and (value = PO) do
begin
  case token of
    thentr: if _evalrule (trule) = TR then value := TR;
    thenfa: if _evalrule (trule) = TR then value := FA;
    elsetr: if _evalrule (trule) = FA then value := TR;
    elsefa: if _evalrule (trule) = FA then value := FA;
    ifpotr: if _evalrule (trule) = PO then value := TR;
    ifpofa: if _evalrule (trule) = PO then value := FA;
  end {case};
  token := _TBSTORE [ptr]; ptr := ptr + 1;
  trule := _TBSTORE [ptr]; ptr := ptr + 1;
end;
```

4. The conclusion is now TR, FA or still PO; assign this value to the rule's conclusion and to the function.

5. Execute the appropriate THELSEs of the rule, if any, by calling procedure _thelse.

6. Take the rule's conclusion and exit.

The only major still unexplained function is _evalexp, the function that evaluates an expression. It is structured as the other procedures and functions. It main body is:

```
ptr := _EVINDEX [rule];
token := _EVSTORE [ptr]; ptr := ptr + 1;
case token of
  codeNOT:     _applyNOT;
  codeMUST:      _applyMUST;
  codePOSS:      _applyPOSS;
  codeAND:     _applyAND;
  codeUCAND:   _applyUCAND;
  codeOR:      _applyOR;
  codeUCOR:      _applyUCOR;
  codeALT:     _applyALT;
  codeHIST:      _applyHIST;
  1..N:        value := _evalrule (token);
  else         _fatal_error ('invalid token in expression')
end {case};
if value = UD then fatal_error ('evalexp returned UD');
_evalexp := value
```

As an example of one of the procedures called by _evalexp, we give the Pascal code for procedure _applyAND, which basically implements a lookup table combined with one or two (recursive) calls to _evalexp:

```
procedure _applyAND;  {computes value, a global variable}
begin
  value := _evalexp;         {evaluate the first argument}
  case value of
    FA: _skipexpr;  {second argument needs no evaluation}
    PO: case _evalexp of  {evaluate second argument, too}
          FA: value := FA; {TR and PO: leave value as is}
        end {case};
    TR: value := _evalexp; {take second argument's value}
  end {case}
end;
```

Procedure _skipexpr skips an expression (the second operand) by positioning ptr, the pointer into the evaluations storage array _EVSTORE, past the tokens that comprise the expression.

From the above it is clear, that all major inferencing procedures and functions are highly (mutually) recursive. This 'recursive descent' evaluation makes the Inference Engine so simple.

References[1]

Aa JJ van der. Intelligent alarms in anesthesia: a real time expert system application. PhD thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1990.

Abkoude WC van. Software for the analysis of the respiratory oxygen signal (in Dutch). BSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1981.

Ackley D, Hinton G, Sejnowski, T. A learning algorithm for Boltzmann machines. Cognitive Science 9, 147-169. 1985.

Allen JF. Maintaining knowledge about temporal intervals. Comm ACM 26:11, 832-843. 1983.

Andre P, Bachy JL, Col J. Computer control of mean arterial pressure with sodium nitroprusside: an adaptive model-based system. Computers in cardiology, Linkoping, Sweden, 471-474. 1985.

Arnsparger JM, McInnis BC, Glover JR, Normann NA. Adaptive control of blood pressure. IEEE Trans Biomed Engng 30:3, 168-176. 1983.

Attinger EO. Parsimonious systems description: a necessary first step in the development of predictive indicators. In: Carson ER, Cramp DG (eds). Computers and control in clinical medicine, 175-212. Plenum. 1985.

Barr A, Feigenbaum EA (eds). The Handbook of Artificial Intelligence, Vols 1, 2 and 3. Kaufman, Los Altos, Calif. 1981.

Bendixen HH, Duberman SM. Decision making in the operating room. In: Grundy BL, Gravenstein JS (eds). The quality of care in anesthesia, 88-99. Springfield, Illinois. 1982.

Beneken JEW, Blom JA, Jorritsma FF, Nandorff A, Spierdijk J. Trend prediction as a basis for optimal therapy. Dept of Electr Engng, Eindhoven Univ of Technology, TH-Report 78-E-86. 1978.

Beneken JEW, Blom JA, Jorritsma FF, Nandorff A, Bijnen A van, Spierdijk J. Servo-anesthesia: model-based prediction and optimal therapy in patients under anesthesia. Biomed Technik 24, Ergänzungsband, 233-234. 1979.

[1] Verslagen van stageairs worden hier BSc thesis genoemd, verslagen van afstudeerders MSc thesis, en rapporten van AIO's (assistenten in opleiding) AIO thesis.

Beneken JEW, Blom JA, Saranummi N. Trends in monitored variables. Proc Symp Control systems concepts and approaches in clinical medicine, 64-67. Brighton. 1982.

Beneken JEW, Blom JA. An integrative patient monitoring approach. In: Gravenstein JS, Newbower RS, Ream AK, Smith NT (eds). An integrated approach to monitoring, 121-131. Butterworths, Boston. 1983.

Beneken JEW, Blom JA, Saranummi N. Accuracy in trend detection. In: Gravenstein JS, Newbower RS, Ream AK, Smith NT (eds). An integrated approach to monitoring, 133-144. Butterworths, Boston. 1983.

Beneken JEW, Blom JA, Meijler AP, Cluitmans P, Spierdijk J, Nandorff A, Nijhuis R, Kessel HM van. Computerized data acquisition and display in anesthesia. In: Prakash O (ed). Computing in anesthesia and intensive care, 25-43. Nijhoff. 1983.

Beneken JEW, Gravenstein JS. Sophisticated alarms: a methodology based on systems engineering concepts. In: Gravenstein JS, Newbower RS, Ream AK, Smith NT (eds). The automated anesthesia record and alarm systems, 211-228. Butterworths, Boston. 1987.

Beneken JEW, van der Aa JJ. Alarms and their limits in monitoring. J Clin Mon 5, 205-210. 1989.

Bierens EJJ. Preliminary study for an expert system based blood pressure controller. MSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1987.

Blom JA, Aa J van der, Jorritsma FF, Beneken JEW, Nandorff A, Spierdijk J, Bijnen A van. A research oriented microcomputer based patient monitoring system. Biomed Technik 26, 135-140. 1981.

Blom JA, Beneken JEW, Jorritsma FF, Gieles JPM, Nandorff A, Spierdijk J. Erkennung von Trends und Fehlern bei Signalen von Patienten unter Narkose. Proc Symp Rechnergestützte Intensivpflege Tuebingen 1979, 126-130. Thieme. 1981.

Blom JA, Beneken JEW. On-line information and data reduction in patient monitoring. In: Paul JP, Jordan MM, Ferguson-Pell MW, Andrews BJ (eds). Computing in Medicine, 117-125. MacMillan. 1982.

Blom JA, Bruijn NP de. Peroperative estimation of sodium nitroprusside sensitivity. Proc IEEE Southeastcon, Sandestin, Fla, 564-566. 1982.

Blom JA, Ruyter JAF de, Saranummi N, Beneken JEW. Detection of trends in monitored variables. In: Carson ER, Cramp DG (eds). Computers and control in clinical medicine, 153-174. Plenum. 1985.

Bobrow DG, Mittal S, Stefik MJ. Expert systems: perils and promise. Commun ACM 29:9. 1986.

Bonissone PP, Tong RM. Reasoning with uncertainty in expert systems. Int J Man-Machine Studies 22, 241-250. 1985.

Boon PMG. Efficiency and correctness of SIMPLEXYS expert systems. MSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1987.

Brachman RJ. What IS-A is and isn't: an analysis of taxonomic links in semantic networks. IEEE Computer 16:10, 30-36. 1983.

Brachman RJ, Levesque HJ (eds). Readings in Knowledge Representation. Kaufmann, Los Altos, Calif. 1985.

Brok MWNM den. A rule based adaptive blood pressure controller (in Dutch). MSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1986.

Brok NWNM den, Blom JA. A rule based adaptive blood pressure controller. Proc 2nd European workshop on fault detection and reliability; knowledge-based and other approaches, Manchester, 67-74. Pergamon Press, Oxford. 1987.

Buchanan BG, Shortliffe EH (eds). Rule-based expert systems; the MYCIN experiments of the Stanford Heuristic Programming Project. Addison-Wesley. 1984.

Burke D. Diagnosis: art or science? Diagn Med 1, 25. 1978.

Butler AR. Further investigations regarding the toxicity of sodium nitroprusside. Clinical chemistry 33:4, 490-492. 1987.

Cantor N, Kihlstrom JF. Social intelligence, the cognitive basis of personality. In: Sage SP (ed). Review of personality and social psychology, vol. 6. Beverley Hills. 1985.

Chellaas BF. Modal logic: an introduction. Cambridge University Press, Cambridge. 1980.

Chizeck HJ. Adaptive control theory and applications to drug delivery. 1986 American Control Conf, Vol 2, 871-873. 1986.

Coad NR. Beta blockade and nitroprusside. Anesthesia 42:9, 1022-1023. 1987.

Cobelli C, Romanin-Jacur G. Controllability, observability and structural identifiability of multi input and multi output biological compartmental systems. IEEE Trans Biomed Engng 23, 93-100. 1976.

Coolegem KG. Processing of the in- and expiratory oxygen signal measurement (in Dutch). BSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1981.

Coolen J. Improvements in the man-machine interaction of a patient monitoring system (in Dutch). MSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1985.

Coombs MJ. Developments in expert systems. In: Hasting DW (ed). Strategic explanations for a diagnostic consultation system. Jovanovich, London. 1984.

Cooper JB, Newbower RS, Long CD. Human error in anesthesia management. In: Gravenstein JS, Paulus DA (eds). Monitoring practice in clinical anesthesia. Lippincott, Philadelphia. 1982.

Cooper JB, Couvillon LA. Accidental breathing system disconnections, Interim report to the FDA. Arthur D Little Inc, Cambridge. 1983.

Davis R. Applications of meta level knowledge to the construction, maintenance and use of large knowledge bases. Report STAN-CS-76-552, Stanford AI Lab, Stanford Univ, Stanford, Calif. July 1976.

Davis R, Buchanan B, Shortliffe EH. Production rules as a representation for a knowledge-based consultation program. Artif Intell 8, 15-45. 1977.

Dijkstra EW. The humble programmer. Comm ACM 15, 859-866. 1972.

Divers RT. The quality of physiologic data for automated records and alarm systems. In: Gravenstein JS, Newbower RS, Ream AK, Smith NT (eds). The automated anesthesia record and alarm systems. Butterworths, Boston. 1987.

Donabedian A. Needed research in the assessment and monitoring of the quality of medical care. NCHSR Research Report Series, HEW Publication (PHS) 78-3219, July 1978.

Dreyfus HL. From micro-worlds to knowledge representation: AI at an impasse. In: Haugeland J (ed). Mind design, 161-204. MIT Press. 1981.

Dreyfus H, Dreyfus S. Why expert systems do not exhibit expertise. IEEE Expert 1, 86-90. Summer 1986.

Ernst GW, Newell A. GPS: a case study in generality and problem solving. ACM monograph, Academic Press, New York. 1969.

Fagan LM. Ventilator manager: a program to provide on-line consultative advice in the intensive care unit. Report HPP-78-16, Computer Science Dept, Stanford University, Stanford, Calif. Sept 1978.

Fagan LM, Kunz JC, Feigenbaum EA. Representation of dynamic clinical knowledge: measurement interpretation in the intensive care unit. Proc IJCAI-79, 260-262. 1979.

Fagan LM. VM: representing time-dependent relations in a medical setting. PhD thesis, Dept of Computer Science, Stanford Univ, Stanford, Calif. June 1980.

Feinstein AR. An analysis of diagnostic reasoning. I. The domains and disorders of clinical macrobiology. Yale J Biol Med 46, 212. 1973.

Feinstein AR. An analysis of diagnostic reasoning. II. The strategy of intermediate decisions. Yale J Biol Med 46, 264. 1973.

Feinstein AR. An analysis of diagnostic reasoning. III. The construction of clinical algorithms. Yale J Biol Med 47, 5. 1974.

Feinstein AR. Clinical bio-statistics XXIX, the Haze of Bayes, the aerial palaces of decision analysis and the computerized Ouija board. Clin Pharmacol Ther 21, 483. 1977.

Feldbrugge F, Jensen K. Petri Net Tool Overview. In: Rozenberg G (ed). Advances in Petri Nets 1986, part 2. Lecture notes in Computer Science, Vol 255, 20-61. Springer, Berlin. 1986.

Feltovich PJ, Barrows HS. Issues of generality in medical problem solving. In: Schmidt HG, Volder ML de (eds). Tutorials in problem-based learning. Van Gorcum, Assen. 1984.

Fink DJ, Galen RS. Probabilistic approaches to clinical decision support. In: Williams BT (ed). Computer aids to clinical decisions, Vol I. CRC Press. 1982.

Gaines BR. Fuzzy and probability uncertainty logics. Information and control 38, 154-169. 1978.

Genderingen HR van. Some aspects of system identification and control theory in the design of an adaptive blood pressure controller (in Dutch). MSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1984.

Goldstein Jr A, Keats AS. The risk of anesthesia. Anesthesiol 33, 130-143. 1970.

Goossens JJM. Signal validation of patient signals (in Dutch). MSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology 1986.

Gravenstein JS. Introduction. In: Gravenstein JS, Newbower RS, Ream AK, Smith NT, Barden J (eds). Monitoring surgical patients in the operating room. Springfield, Illinois. 1979.

Gupta A, Prasad BE (eds). Principles of expert systems. IEEE Press, New York. 1988.

Hair PJA de. Realization of an explain facility for SIMPLEXYS expert systems. MSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1988.

Hajek J. A knowledge engineering logic for smarter, safer and faster (expert) systems. EUT Computing Centre Note 41, Eindhoven Univ of Technology. 1988.

Hammond JJ, Kirkendall WM, Calfee RV. Hypertensive crisis managed by computer controlled infusion of sodium nitroprusside: a model for the closed loop administration of short acting vasoactive agents. Comp and Biomed Res 12, 97-108. 1979.

Harmon P, King D. Expert Systems. Wiley. 1985.

Hart P. Peter Hart talks about expert systems. IEEE Expert 1, 96-99. Spring 1986.

Hayes-Roth F, Waterman DA, Lenat D (eds). Building Expert Systems, Addison-Wesley. 1983.

Hayes-Roth F. A blackboard architecture for control. Artif Intell, 26:3, 251-321. July 1985.

He WG, Kaufman H, Roy R. Multiple model adaptive control procedure for blood pressure control. IEEE Trans Biomed Engng 33:1, 10-19. 1986.

Hendler JA. Expert systems: the user interface. Ablex, Norwood, NJ. 1988.

Hengst J, Krämer B. A data acquisition system for patient monitoring in anesthesia (in Dutch). MSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1980.

Hill FJ, Peterson GR. Digital systems: hardware organization and design, 2nd ed. Wiley, New York. 1978.

Hofstadter DR. Metamagical themas: Questing for the essence of mind and pattern. Bantam. 1986.

Hoogendoorn P. Automated blood pressure control, a literature review. BSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1988.

Hoogendoorn P. The design of a rule based blood pressure controller. MSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1989.

Hopking BDA. Hazards and errors in anesthesia. Springer. 1980.

Horvitz EJ. Toward a science of expert systems. In: Computer Science and Statistics. Proc 18th Symp on the Interface. American Statistical Assoc, Fort Collins, 45-52. March 1986.

Hughes GE, Cresswell MJ. An introduction to modal logic. Methuen. 1968.

Hutchinson WF, Hollway, TE. An interesting effect of sodium nitroprusside [letter]. Anesthesia 40:11, 1128. 1985.

Jazwinski AH. Stochastic processes annd filtering theory. Academic Press, New York. 1970.

Jensen K. Computer Tools for Construction, Modification and Analysis of Petri Nets. In: Rozenberg G (ed). Advances in Petri Nets 1986, part 1. Lecture notes in Computer Science, Vol 254, 4-19. Springer, Berlin. 1986.

Kanal LN, Lemmer JF. Uncertainty in Artificial Intelligence. North-Holland. 1986.

Kary DD, Juell PL. TRC, an expert system compiler. SIGPLAN Notices 21:5, 64-68. May 1986.

Kelly J. Intelligent machines, what chance? In: Hallam J, Mellish C (eds). Advances in Artificial Intelligence. Wiley, Chichester. 1987.

Kessel HM van. Design of a data acquisition and display system for patient monitoring in anesthesia (in Dutch). MSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1981.

Khoroshevsky VF. ATN-based explanation subsystems: design and implementation. Comp and Artif Intell 4:4, 289-311. 1985.

Klingler DE. Rapid prototyping revisited. Datamation 32:20, 131-132. 1986.

Krijgsman AJ, Verbruggen HB, Bruijn PM. Knowledge based real time control. In: Rodd MG, Suski GJ (eds). Artificial Intelligence in real-time control. Pergamon. 1989.

Laffey TA, Cox PA, Schmidt JL, Kao SM, Read JY. Real time Knowledge-Based Systems. AI Magazine, 27-45, Spring 1988.

Lammers JO. The use of Petri net theory for SIMPLEXYS expert systems protocol checking. AIO thesis, Dept of Electr Engng, Eindhoven Univ of Technology. EUT-Report 90-E-238. 1990.

Lammers JO. Knowledge based adaptive blood pressure control: a SIMPLEXYS expert system application. AIO thesis, Dept of Electr Engng, Eindhoven Univ of Technology. EUT-Report 90-E-236. 1990.

Lenat DB. The nature of heuristics. Artif Intell 19, 189-249. 1982.

Lutgens JMA. A tautology checker for the SIMPLEXYS expert system (in Dutch). BSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1989.

Mamdani EH, Efstathiou J. An analysis of formal logics as inference mechanisms in expert systems. In: Gupta A, Prasad BE (eds). Principles of expert systems. IEEE Press, New York. 1988.

Manteleers JJH. Critical incidents in anesthesia, a literature survey. BSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1985.

Marsh J, Greenwood J. Guide to defense and aerospace systems. Pasha Publ, Arlington, Va. 1986.

Martin JF, Schneider AM, Smith NT. Multiple model adaptive control of blood pressure using sodium nitroprusside. IEEE Trans Biomed Engng 34:8, 617-623. 1987.

McInnis BC, Deng LZ. Automatic control of blood pressures with multiple drug inputs. Ann of Biomed Engng 13, 217-225. 1985.

McNally RT, Engelman K, Noordergraaf A, Edwards Jr M. A device for the precise regulation of blood pressure in patients during surgery and hypertensive crises. Proc San Diego Biomed Symp 16, 419-424. 1977.

Medawar PB. Induction and intuition in scientific thought. American Philosophical Soc, Philadelphia. 1969.

Meijler AP. Automation in anesthesia, a relief? PhD thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1986. Springer, 1987.

Meijler AP, Beneken JEW. Data acquisition and display: a system with centralized display, automatic record keeping, and intelligent alarms. In: Gravenstein JS, Newbower RS, Ream AK, Smith NT (eds). The automated anesthesia record and alarm systems. Butterworths, Boston. 1987.

Meline LJ, Westenskow DR, Somerville A, Wernick RT, Pace NL. Evaluation of two adaptive SNP control algorithms. J Clin Monit 2:2, 79-86. 1986.

Melle W van, Shortliffe EH, Buchanan BG. EMYCIN, a domain-independent system that aids in constructing knowledge-based consultation programs. In: Bond A (ed). Machine Intelligence, Infotech State of the Art Report, Series 9 No 3, Pergamon Infotech Ltd. 1981.

Meltzer B. The programming of Deduction and Induction. In: Elithorn A, Jones D (eds). Artificial and Human Thinking. Elsevier. 1973.

Michie D. Expert Systems. The Computer Journal 23:4, 371. 1980.

Millard RK, Hutton P, Pereira E, Prys-Roberts C. On using a self-tuning controller for blood pressure regulation during surgery in man. Comp Biol and Med 17:1, 1-18. 1987.

Milne RW. A few problems with expert systems. In: Gupta A, Prasad BE (eds). Principles of expert systems. IEEE Press, New York. 1988.

Mitchell MM. Human factors in the man-machine interface. In: Gravenstein JS, Newbower RS, Ream AK, Smith NT (eds). The automated anesthesia record and alarm systems. Butterworths, Boston. 1987.

Murphy EA. The logic of medicine. John Hopkins University Press, Baltimore. 1976.

Neapolitan RE. Forward-chaining versus a graph approach as the inference engine in expert systems. In: Gupta A, Prasad BE (eds). Principles of expert systems. IEEE Press, New York. 1988.

Nelson PE, Ream AK. Monitor calibration for an automated anesthesia record. In: Gravenstein JS, Newbower RS, Ream AK, Smith NT (eds). The automated anesthesia record and alarm systems. Butterworths, Boston. 1987.

Newell A, Simon HA. Human Problem Solving. Prentice-Hall, Englewood Cliffs, NJ. 1972.

Newell A, Rosenbloom PS. Mechanisms of skill acquisition and the law of practice. In: Anderson JR (ed). Cognitive skills and their acquisition. Erlbaum, Hillsdale, NJ. 1981.

Newell A. Foreword. In: Buchanan BG, Shortliffe EH (eds). Rule-based expert systems; the MYCIN experiments of the Stanford Heuristic Programming Project. Addison-Wesley. 1984.

Nguyen TA. Verifying consistency of production systems. In: Gupta A, Prasad BE (eds). Principles of expert systems. IEEE Press, New York. 1988.

Nilsson NJ. Problem-solving methods in Artificial Intelligence. McGraw-Hill. 1971.

Niwa K, Sasaki K, Ihara H. An experimental comparison of knowledge representation schemes. AI Mag, 29-36, Summer 1984.

O'Reilly CA, Cromarty AS. 'Fast' is not 'Real time' in designing effective real time AI systems. In: Appl of Artif Intell II, 249-257. Int Soc of Optical Engng, Bellingham, Wash. 1985.

Osterweil L. Integrating the testing, analysis and debugging of programs. Proc Symp Software Validation, Darmstadt, FRG, 73-102. September 1983.

Pace NL, Westenskow DR: Computer regulated sodium nitroprusside infusion for blood pressure control. In: Prakash O (ed). Computing in Anesthesia and Intensive Care, 292-301. Nijhoff. 1983.

Packer JS, Mason DG, Cade JF, McKinley SM. An adaptive controller for closed-loop management of blood pressure in seriously ill patients. IEEE Trans Biomed Engng 34:8, 612-616. 1987.

Pass TM, Komaroff AL, Ervin CT. Categorical approaches to clinical decision support. In: Williams BT (ed). Computer aids to clinical decisions, Vol I, 95-138. CRC Press. 1982.

Patel CB, Laboy V, Venus B, Mathru M, Wier D. Use of sodium nitroprusside in post-coronary bypass surgery; a plead for conservatism. Chest 89:5, 663-667. 1986.

Pau LF. Prototyping, validation and maintenance of knowledge based systems software. In: Third Annual Proc of expert systems in Government, Washington. October 1987.

Pearl J. Fusion, propagation, and structuring in Bayesian networks. Symp on complexity of approximately solved problems, Columbia University. 1985.

Peters MMJ. Optimization of the trend detection algorithm (in Dutch). MSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1984.

Petri CA. Kommunikation mit Automaten. Schriften des Institutes für Instrumentelle Mathematik, Bonn. 1962.

Petri CA. Concurrency Theory. In: Rozenberg G (ed). Advances in Petri Nets 1986, part 1. Lecture notes in Computer Science, Vol 254, 4-24. Springer, Berlin. 1986.

Philippens EHJ. Designing debugging tools for SIMPLEXYS expert systems. MSc thesis, Dept of Electr Engng, Eindhoven University of Technology. December 1989.

Plasman JLC. Analysis of the arterial and venous blood pressure signals (in Dutch). MSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1981.

Politakis P, Weiss SM. Using empirical analysis to refine expert system knowledge bases. Artif Intell 22, 23-48. 1984.

Post E. Formal reductions of the general combinatorial problem. Am J of Math 65: 197-268. 1943.

Pratt JW, Raiffa H, Schlaifer R. Introduction to statistical decision theory. McGraw-Hill, New York. 1965.

Quillian MR. Word concepts: a theory and simulation of some basic semantic capabilities. Behavioral Science 12, 410-430. 1967.

Quine W Van Orman. Methods of logic. Routledge and Kegan Paul. 1958.

Rademakers MAJ, Schelle HJ. Capnography and its automatic diagnostics (in Dutch). BSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1983.

Raiffa H. Decision Analysis: Introductory lectures on choice under uncertainty. Addison-Wesley, Reading, Mass. 1968.

Rampil IJ. Intelligent detection of artifact. In: Gravenstein JS, Newbower RS, Ream AK, Smith NT (eds). The automated anesthesia record and alarm systems. Butterworths, Boston. 1987.

Ream AK. Monitoring concepts and techniques. In: Ream AK, Fogdall RP (eds). Acute cardiovascular management; anesthesia and intensive care, 139-160. Lippincott. 1982.

Ream AK, Robinson DJ, Nelson PE, Portner PM. The automated anesthesia record: software design philosophy. In: Gravenstein JS, Newbower RS, Ream AK, Smith NT (eds). The automated anesthesia record and alarm systems. Butterworths, Boston. 1987.

Reid JA, Kenny GNC. Evaluation of closed loop control of arterial pressure after cardiopulmonary bypass. Br J of Anaesthesia 59:2, 247-255. 1987.

Reisig W. Petri Nets. EATCS Monographs on Theoretical Computer Science, Vol 4. Springer, Berlin. 1985.

Rich E. Artificial intelligence. McGraw-Hill. 1983.

Richer MH. An evaluation of expert system development tools. Expert Syst 3:3, 166-183. 1986.

Rodd MG, Suski GJ (eds). Artificial Intelligence in real-time control. Pergamon. 1989.

Rosenbloom PS. The chunking of goal hierarchies: a model of practice and stimulus-response compatibility. PhD thesis, Carnegie-Mellon Univ. 1983.

Rosenbloom PS, Newell A. The chunking of goal hierarchies: a generalized model of practice. In: Michalski RS, Carbonell JG, Mitchell TM (eds). Machine learning, an Artificial Intelligence approach, Vol 2, 247-288. Kaufmann, Los Altos. 1986.

Rosenfeld I. The complete medical exam. Simon and Schuster, New York. 1978.

Rosenfeldt FL, Chang V, Grigg H, Parker S, Learns R, Rabinov M, Xu WG. A closed loop microprocessor controller for treatment of hypertension after cardiac surgery. Anesth and Intensive Care 14:2, 158-162. 1986.

Russek E, Kronmal RA, Fisher LD. The effect of assuming independence in applying Bayes' theorem to risk estimation and classification in diagnosis. Comp and Biomed Res 16:6, 537-552, Dec 1983.

Ruyter JAF de. Trend detection in physiological signals (in Dutch). MSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1982.

Schneider AJL. The current practice. In: Gravenstein JS, Newbower RS, Ream AK, Smith NT, Barden J (eds). Monitoring surgical patients in the operating room, 5-16. Springfield, Illinois. 1979.

Schoor BFW. The trend detection algorithm as a basis for an intelligent alarms expert system (in Dutch). MSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1986.

Shafer G. A mathematical theory of evidence. Princeton University Press. 1976.

Sheppard LC, Kouchoukos NT, Shotts JF, Wallace FD. Regulation of mean arterial pressure by computer control of vasoactive agents in postoperative patients. Proc Computers in Cardiology, 91-94. 1975.

Sheppard LC. Correlation analysis of blood pressure responses to vasoactive drugs, with particular reference to clinical surveillance of the post-surgical cardiac patient. PhD thesis, Imperial College, London. 1976.

Shore JE. Relative entropy, probabilistic inference and AI. In: Uncertainty in Artificial Intelligence, North Holland. 1986.

Shortliffe EH, Buchanan BG. A model of inexact reasoning in medicine. Math Biosciences 23, 351-379. 1975.

Shortliffe EH. Computer-based medical consultations, MYCIN. Elsevier, New York. 1976.

Shortliffe EH, Fagan LM. Expert systems research: modeling the medical decision making process. Technical Memo HPP-82-3, Computer Science Dept, Stanford Univ, Stanford, Calif. 1982.

Shuman L. Evaluation of the process of anesthesia care. In: Grundy BL, Gravenstein JS (eds). The quality of care in anesthesia, 61-87. Springfield, Illinois. 1982.

Simmonds WH. Representation of real knowledge for real time use. In: Rodd MG, Suski GJ (eds). Artificial Intelligence in real-time control. Pergamon. 1989.

Simon HA. The theory of problem solving. Information Processing 71, 261-277. 1972.

Skolnick M: The impact of technology on anesthesia care: computers, communications, and the neurosciences. In: Grundy BL, Gravenstein JS (eds). The quality of care in anesthesia. Springfield, Illinois. 1982.

Slate JB. Model based design of a controller for infusing sodium nitroprusside during postsurgical hypertension. PhD thesis, Univ of Wisconsin, Madison. 1980.

Slate JB, Sheppard LC. Automatic control of blood pressure by drug infusion. Proc IEE 129:9, 639-644. 1982.

Stern KS, Chizeck HJ, Walker BK, Krishnaprasad PS, Katona PG. The self tuning controller, comparison with human performance in the control of arterial pressure. Ann of Biomed Engng 13, 341-347. 1985.

Suwa M, Scott C, Shortliffe EH. Completeness and consistency in a rule-based system. In: Buchanan BG and E.H. Shortliffe EH (eds). Rule-based expert systems, 159-170. Addison-Wesley. 1984.

Szolovits P, Pauker SG. Categorical and probabilistic reasoning in medical diagnosis. Artif Intell 11, 115. 1978.

Tsang EPK. TLP, a temporal planner. In: Hallam J, Mellish C (eds). Advances in Artificial Intelligence, 63-80. Wiley, Chichester. 1987.

Tullemans MH. Expert systems and the implementation of expertise; a cognitive analysis of the human expert (in Dutch). BSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1987.

Turner M. Real Time Experts. Systems Int 14, 1, 55-57. 1986.

Vetter NJ, Julian DG. Comparison of arrhythmia computer and conventional monitoring in a coronary care unit. The Lancet, 1151-1154. 1975.

Vingerhoeds RA, Delbar P, Boullart L. Expert systems for process control using automatic knowledge acquisition. In: Rodd MG, Suski GJ (eds). Artificial Intelligence in real-time control. Pergamon. 1989.

Voss GI, Katona PG, Chizeck HJ. Automated control of arterial blood pressure and cardiac output with sodium nitroprusside and dobutamine in anesthetized dogs. 1986 American Control Conf, Vol 2, 874-877. 1986.

Waterman DA. A Guide to Expert Systems, Addison-Wesley. 1986.

Weed LL. Medical records, medical education and patient care. Yearbook Medical, Chicago. 1971.

Westenskow DR, Meline L, Pace NL. Controlled hypotension with sodium nitroprusside: anesthesiologist versus computer. J Clin Mon 3:2, 80-86. 1987.

Winograd T. Understanding natural language. Academic Press. 1972.

Winston PH. Artificial Intelligence, Addison-Wesley. 1977.

Wirth N. Programming languages: what to demand and how to assess them. Institut für Informatik, report nr 17. ETH Zürich. 1976.

Wirth N. Algorithms + Data Structures = Programs. Prentice Hall, Englewood Cliffs, NJ. 1976.

Wood M, Hyman S, Wood AJJ. A clinical study of sensitivity to sodium nitroprusside during controlled hypotensive anesthesia in young and elderly patients. Anesthesia and Analgesia 66:2, 132-136. 1987.

Woods WA. Important issues in knowledge representation. Proc IEEE 74:10, 1322-1334. 1986.

Woord H van der. Characteristics of sodium nitroprusside infusion for the design of an adaptive blood pressure controller. MSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1981.

Zwart R. Implementation and evaluation of a robust adaptive blood pressure controller (in Dutch). MSc thesis, Dept of Electr Engng, Eindhoven Univ of Technology. 1990.

Summary

This dissertation describes the SIMPLEXYS toolbox, a collection of tools to design real time expert systems. The central tool is a new expert systems language that is especially meant to formulate and solve problems in the domain of patient monitoring and clinical control systems.

SIMPLEXYS expert systems are more *efficient* and *safe* than many other expert systems. Their efficiency is caused by the fact that the SIMPLEXYS Inference Engine is based on a *linear time* algorithm. During the inferencing process, *no searching is necessary*; searching is done by a compiler, which converts the knowledge base into an internal format which can be executed at high speed. This is essential in real time expert systems. The safety of SIMPLEXYS expert systems originates in the numerous ways in which the *correctness of the knowledge base* can be checked; this, too, is due to the fact that the knowledge base is compiled. The compiler creates a dynamic and a static network, the properties of which can be analyzed against a variety of criteria that must be satisfied in a correct knowledge base.

Basic features of the SIMPLEXYS language are its implementation of *goals*, which describe what must be done, and of *protocols*, which describe when to do what. Protocols denote a *context*, and the context specifies which of the goals must be pursued. Protocols can efficienty provide an answer to many of the questions that temporal logics are designed to answer.

Properties of the SIMPLEXYS language are 1) that it allows a knowledge base designer to clearly and concisely describe both symbolic and procedural knowledge and the ways in which these interact, 2) that it allows the knowledge to be translated into an internal format which can be rapidly executed by even a small computer, 3) that the correctness of the knowledge can be tested in a variety of ways, and 4) that it provides convenient and efficient methods to perform calculations and to interface with the 'outside world'.

The basic unit of the language is the *rule*. SIMPLEXYS rules do not have the familiar 'if ... then ...' format that is used in production systems but a definitional one, '... is defined as ...'. From this, and from the three-valued logic that is used in SIMPLEXYS, follows another major difference with production systems: a SIMPLEXYS rule has as its default conclusion *unknown* rather than *false*, other possible conclusions being *true* and *false*. Another difference is that a SIMPLEXYS rule can lead to multiple conclusions. The combination of these properties leads to the possibility to check for a number of inconsistency errors in the knowledge base; many of these checks can be performed at compile time.

Other SIMPLEXYS tools are, besides the language: the Rule Compiler and its extensions, which translate and check the knowledge, the Inference Engine which provides

the reasoning ability and the Tracer/Debugger, which can analyze the inferencing that took place during an expert system's operation.

Chapter 1 gives a general description of the goal of the SIMPLEXYS experiment: to provide a set of tools to solve a particular class of problems in patient monitoring.

Chapter 2 provides some background on expert systems and reviews in particular the requirements that must be met and the problems that are encountered in real time expert system applications; specific problems are the slow execution speed of rule-based systems due to the time spent in searching, and the unpredictability of the system's response time.

Chapter 3 provides a background on problem solving in medicine, and particularly in patient monitoring in anesthesia. SIMPLEXYS formalizes the *hypothetico-deductive problem solving* approach and the models that describe diagnostic and therapeutic management which are known as *protocols*. The link with machine reasoning and with the SIMPLEXYS problem solving methodology is demonstrated. The chapter concludes with some issues that are important in the design of a new programming language, which is to be a tool to build a logical, verifiable 'abstract machine' to solve a certain class of problems given a convenient and appropriate set of building blocks.

Chapter 4 provides the necessary insight into the features that the SIMPLEXYS language provides and gives a motivation for their existence. It stresses the needs for simplicity and clarity of the language's constructs, efficiency in their evaluation, and appropriateness of the constructs within the problem domain.

Next, chapter 5 gives a detailed description of all SIMPLEXYS constructs and how they are used, describes the various tools, and gives some insight in how efficiently SIMPLEXYS expert systems perform. In particular, the SIMPLEXYS inferencing mechanism is shown to be approximately linear time. One of the causes for this is the fact that no searching need be done by the inferencing mechanism.

Chapters 6 and 7 describe in detail how the semantics of the knowledge base can be checked. Checking is feasible because the rules, the elementary chunks of knowledge, are connected into a logical network that must have certain pre-specified properties. Chapter 6 considers the correctness of the static aspects of the knowledge, e.g. contradictions and circular definitions, and chapter 7 its dynamic aspects, e.g. whether deadlock can occur.

Chapter 8 considers what to do when real time expert systems must process large quantities of rapidly changing data. Data processing *algorithms* can compact the data by eliminating redundancies and computing symbolic 'features' that the expert system will be able to handle at a symbolic level. Data can also contain *artifacts*, which must be detected; otherwise the expert system might reach erroneous conclusions based on corrupted

information. Besides some issues concerning data input and data processing, this chapter presents a *feature extraction and validation* methodology for physiological signals.

Chapter 9 describes one of the SIMPLEXYS applications, a sodium nitroprusside blood pressure controller, whose task it is to stabilize a patient's mean arterial pressure at a lower than normal level. This chapter reviews the available knowledge about sodium nitroprusside, sodium nitroprusside controllers, especially those of the PID type, and the adaptation of the controller's properties which is necessary due to the large sensitivity range in patients. Some implementation details are described, which demonstrate how protocols are used to specify expectations. The chapter concludes with a review of the performance of the controller and some of the remarks of the knowledge engineer about the role of SIMPLEXYS in its design.

Chapter 10, finally, concludes that SIMPLEXYS provides an appropriate set of tools for the design of high performance real time experts in the patient monitoring domain.

Samenvatting

Deze dissertatie beschrijft SIMPLEXYS, een verzameling gereedschappen om real time expert systemen te ontwerpen en te ontwikkelen. Het centrale gereedschap is een nieuwe expert systeem programmeertaal die in het bijzonder is bedoeld om problemen op het gebied van patientbewaking en klinische regelsystemen te formuleren en op te lossen.

SIMPLEXYS expert systemen zijn *sneller* en *veiliger* dan vele andere. Hun snelheid wordt veroorzaakt door het feit dat de Inference Engine van SIMPLEXYS gebaseerd is op een algoritme dat slechts *lineair met de tijd* is. Tijdens het inferentieproces is *zoeken niet nodig*; zoeken wordt gedaan door een compiler, die het kennisbestand converteert naar een interne vorm die met hoge snelheid kan worden verwerkt. Dit is essentiëel in real time expert systemen. De veiligheid van SIMPLEXYS expert systemen berust op de vele wijzen waarop de *correctheid van het kennisbestand* gecontroleerd kan worden; ook dit is mogelijk vanwege het feit dat het kennisbestand wordt gecompileerd. De compiler creëert een dynamisch en een statisch netwerk, waarvan de eigenschappen kunnen worden geanalyseerd ten aanzien van een veelheid van criteria waaraan voldaan moet zijn wil het kennisbestand correct zijn.

Basis-aspecten van de SIMPLEXYS taal zijn de implementatie van *goals* (doelstellingen), die beschrijven *wat* er gedaan moet worden, en van *protocols*, die beschrijven *wanneer* iets gedaan moet worden. Protocols duiden een *context* aan, die specificeert welke van de doelstellingen gevolgd moet worden. Protocols kunnen op efficiente wijze antwoord geven op de meeste vragen die een temporele logica kan beantwoorden.

Eigenschappen van de SIMPLEXYS taal zijn 1) dat zij de expert systeem ontwerper in staat stelt zowel symbolische als procedurele kennis als ook de wijze waarop deze samenhangt duidelijk en beknopt weer te geven, 2) dat de kennis vertaald kan worden naar een interne representatie die snel kan worden gemanipuleerd door zelfs een kleine computer, 3) dat de correctheid van de kennis op een aantal verschillende manieren gecontroleerd kan worden, en 4) dat zij geschikte en efficiente methoden biedt om berekeningen uit te voeren en om met de 'buitenwereld' te communiceren.

De basis-eenheid van de taal is de *regel*. Regels hebben in SIMPLEXYS niet de bekende 'als ... dan ...' vorm van de productie-systemen, maar de vorm van een definitie, '... is gedefiniëerd als ...'. Hieruit, en uit de driewaardige logica die in SIMPLEXYS wordt gebruikt, volgt nog een belangrijk verschil met de productie-systemen: een conclusie heeft in SIMPLEXYS de waarde *onbekend* indien voor die conclusie noch *waar*, noch *onwaar* kan worden afgeleid; in productie-systemen is een conclusie *onwaar* tenzij *waar* kan worden afgeleid. Nog een verschil is dat in SIMPLEXYS uit één regel meerdere conclusies kunnen volgen. Deze eigenschappen leiden tesamen tot de mogelijkheid het kennisbestand op een aantal mogelijke vormen van inconsistentie te controleren; veel van deze controles kunnen reeds worden uitgevoerd bij het vertalen van de kennis naar zijn interne representatie.

Naast de taal zijn andere gereedschappen die SIMPLEXYS aanbiedt: de 'Rule Compiler' en zijn extensies, die de kennis vertalen en controleren, de 'Inference Engine' die het redeneermechanisme incorporeert, en de 'Tracer/Debugger', die het redeneerproces zoals dat plaats vond tijdens het in bedrijf zijn van een expert systeem kan analyseren.

Hoofdstuk 1 geeft een algemene beschrijving van de doelstelling van het SIMPLEXYS experiment: het leveren van een aantal gereedschappen om een welbepaalde klasse problemen op het gebied van de patientbewaking op te lossen.

Hoofdstuk 2 biedt enige achtergrondinformatie over expert systemen en beschouwt in het bijzonder de eisen die gesteld moeten worden en de problemen die tegemoet gezien kunnen worden bij het toepassen van real time expert systemen; in het bijzonder zijn de geringe executiesnelheid van regel-gebaseerde expert systemen, veroorzaakt door de tijd die nodig is voor zoeken, en de onvoorspelbaarheid van de responstijd van het systeem problemen die een oplossing behoeven.

Hoofdstuk 3 beschouwt hoe medische problemen opgelost worden, in het bijzonder in de patientbewaking in de anesthesie. SIMPLEXYS formaliseert de *hypothetico-deductieve methode* van probleem-oplossen en de onder de naam *protocol* bekend staande modellen die het diagnostische en therapeutische handelen beschrijven. Ook wordt het verband gelegd met machinaal redeneren en met de methodologie voor het oplossen van problemen die in SIMPLEXYS wordt gehanteerd. Dit hoofdstuk besluit met enige zaken die van belang zijn bij het ontwerp van een nieuwe programmeertaal, die immers een hulpmiddel moet zijn om een logische, verifieerbare 'abstracte machine' te bouwen die een bepaalde klasse problemen kan oplossen gegeven een aantal handige en toepasselijke bouwstenen.

Hoofdstuk 4 biedt het noodzakelijke inzicht in en een motivatie voor de eigenschappen van de SIMPLEXYS taal. De eenvoud en helderheid van de basis-elementen van de taal, de efficientie van de evaluatie daarvan, en de toepasselijkheid ervan voor het probleem-domein worden benadrukt.

Vervolgens geeft hoofdstuk 5 een gedetailleerde beschrijving van alle basis-elementen van SIMPLEXYS en hoe ze gebruikt worden, het beschrijft de diverse gereedschappen, en het geeft enig inzicht in de executie-snelheid van SIMPLEXYS expert systemen. In het bijzonder wordt aangetoond dat het redeneermechanisme van SIMPLEXYS bij benadering lineair is met de tijd. Dit is onder andere een gevolg van het feit dat het inferentie-mechanisme niet behoeft te zoeken.

Hoofdstukken 6 en 7 beschrijven in detail hoe de semantiek van het kennisbestand gecontroleerd kan worden. Controle is mogelijk omdat de regels, de elementaire kennis-entiteiten, verbonden worden tot een netwerk dat aan bepaalde van te voren gespecificeerde eigenschappen moet voldoen. Hoofdstuk 6 beschouwt de correctheid van de statische

aspecten van de kennis zoals contradicties en circulaire definities, en hoofdstuk 7 de dynamische aspecten zoals de mogelijkheid het einde van het protocol te bereiken.

Hoofdstuk 8 beschouwt hoe gehandeld kan worden indien een real time expert systeem een grote hoeveelheid snel veranderende gegevens moet verwerken. Gegevensverwerkende *algoritmes* kunnen de gegevens samenvatten en symbolische 'eigenschappen' berekenen die het expert systeem vervolgens op symbolisch niveau kan verwerken. In de gegevens kunnen ook *artefacten* aanwezig zijn; deze dienen gedetecteerd te worden omdat het expert systeem anders foutieve conclusies zou kunnen afleiden tengevolge van gecorrumpeerde informatie. Naast enkele zaken betreffende de invoer van gegevens en gegevensverwerking biedt dit hoofdstuk een methodologie om de 'eigenschappen' van fysiologische signalen te extraheren en om die te valideren.

Hoofdstuk 9 beschrijft één van de toepassingen van SIMPLEXYS, een bloeddrukregelaar, waarvan het de taak is door middel van infusie van natrium-nitroprusside de bloeddruk van een patient op een kunstmatig verlaagd niveau te stabiliseren. Dit hoofdstuk geeft een overzicht van de beschikbare kennis op het gebied van natrium-nitroprusside, automatische regelaars die deze stof toedienen, in het bijzonder die van het PID-type, en de adaptatie van de eigenschappen van de regelaar die noodzakelijk is vanwege de grote spreiding van de gevoeligheid in patienten. Enkele implementatie-details, die laten zien hoe protocols gebruikt worden om verwachtingen te specificeren, worden beschreven. Het hoofdstuk besluit met een overzicht van de performance van de regelaar en met enkele van de opmerkingen van de ontwerper van het systeem over de rol die SIMPLEXYS speelde bij het ontwerp.

Tenslotte concludeert hoofdstuk 10 dat SIMPLEXYS een geschikt assortiment hulpmiddelen biedt om real time expert systemen van hoge kwaliteit te kunnen realiseren op het gebied van patientbewaking.

Curriculum Vitae

Johannes Abraham Blom werd geboren op 3 april 1944. Van 1955 tot 1961 bezocht hij het Corderius Lyceum in Amersfoort, waar hij het diploma Gymnasium-ß behaalde. Daarna bracht hij als 'exchange student' een jaar in de Verenigde Staten door, waar hij in 1962 het High School diploma behaalde aan de University High School in West Los Angeles.

Van 1962 tot 1970 studeerde hij aan de afdeling Elektrotechniek van de Technische Hogeschool te Eindhoven. De afstudeerrichting was Meet- en Regeltechniek en zijn afstudeerhoogleraar was prof.ir. D.H. Bekkering. De afstudeeropdracht werd verricht aan het Medisch Fysisch Instituut TNO te Utrecht en betrof de analyse van met oppervlakte-elektroden geregistreerde elektromyografische signalen. In 1970 verkreeg hij het ingenieursdiploma.

Van 1971 tot 1972 was hij werkzaam op het Medisch Fysisch Instituut TNO te Utrecht, waar hij werkte aan de opzet van een deelmodel van het perifere zenuwstelsel dat verschillende verschijnselen in het elektromyogram zou kunnen verklaren.

Als gewetensbezwaarde militaire dienst werd hij in 1972 te werk gesteld bij de vakgroep Meet- en Regeltechniek van de Technische Hogeschool te Eindhoven, waar hij in samenwerking met prof.ir. D.H. Bekkering begon te werken aan de mathematische modellering van en parameterschatting aan fysiologische systemen. Vanwege een tijdelijke aanstelling aan de THE kon hij dit werk tot 1974 voortzetten, nu in samenwerking met prof.dr.ir. J.E.W. Beneken. In deze periode begon zijn werk zich te specialiseren in de richting van anesthesie en patientbewaking.

Van 1974 tot 1975 werkte hij bij het Department of Surgery van de University of Alabama in Birmingham, waar hij, in samenwerking met dr. L.C. Sheppard, zijn werk voortzette, nu gericht op een optimale regeling van bloedinfusie aan patienten, die na een hartoperatie op de intensive care unit verbleven.

In 1975 keerde hij terug naar de vakgroep Meet- en Regeltechniek van de Technische Hogeschool te Eindhoven. Hij werd, onder verantwoordelijkheid van prof.dr.ir. J.E.W. Beneken, leider van het servo-anesthesie projekt, waarvan de doelstelling was en is het onderzoek in hoeverre automatisering in de anesthesie zinvol en mogelijk is. Sinds in 1980 de vakgroep Medische Elektrotechniek gevormd werd, is hij daar werkzaam.

In 1985 begon hij theoretisch en praktisch onderzoek op het gebied van kunstmatige intelligentie en expert systemen. Hij ontwikkelde een nieuwe, speciaal voor taken als patientbewaking bedoelde, expert systeem programmeertaal en de daarbij behorende compiler en andere hulpmiddelen. Op grond van dat werk is deze dissertatie to stand gekomen.

ADDENDUM

to

The SIMPLEXYS Experiment

real time expert systems in patient monitoring

by

J.A. Blom

Ir. Jan Hajek has brought it to my attention, that the thanks that I offer him in my (Dutch) foreword have been misunderstood, by at least one person. Therefore maybe by others, too. This addendum wants to take such an impression away, if it may have arisen. What I meant was most emphatically a compliment. Let me elaborate, and in recompense add some extra praise and acknowledge the indispensable but often invisible role which Jan Hajek played in the design history of the SIMPLEXYS system.

I hope that at least the statement in the foreword, that Hajek put me on the track of what was to become SIMPLEXYS, is unmistakable. This early start has only briefly been mentioned on page 51. Let me add some more details. It came about as follows.

In May 1986 Hajek presented his first Artificial Intelligence course at the Eindhoven University of Technology, in which I was probably the most enthousiast (and most obnoxious?) student. I am sure I graduated, although I did not get a grade. Following this course, upto about May 1987, we became both collaborators and competitors. Jan Hajek was developing his expert system Quixpert, I was working on what was going to be called SIMPLEXYS, and of course both of us wanted to be best. We frequently had long, heated, detailed, in-depth discussions about the features that fast and safe expert systems should have.

How to adequately describe this creative process in action, this battle of minds, in rational, objective, scientific terms? Such a creative process is non-rational, like a championship ping-pong game in which ideas, ripe and unripe, are, like balls, feverishly bounced back and forth in an effort to score. New ideas grow or suddenly erupt. And a great many of my ideas originated, grew or ripened during such intensive exchanges. The term "destructive mind set", mentioned in the foreword, is Jan Hajek's own phrase for his critical attitude and high standards where software in general and expert systems in particular are concerned. He knows exactly what I mean. And this attitude was very helpful to me in discriminating worthwhile ideas from useless ones.

If an idea survived such a confrontation, it probably was a good one. And, stubborn as I am, I did not give up easily at all. That is what the foreword means when it says that his 'destructive mind set' stimulated me to do it anyway.

u \ v	TR	FA	PO
TR	TR	TR	TR
FA	FA	FA	FA
PO	TR	FA	PO

u DEFAULT v

u \ v	TR	FA	PO
TR	TR	**	TR
FA	**	FA	FA
PO	TR	FA	PO

u ALT v

One area of discourse was, that both Jan Hajek and I were convinced that a three-valued logic needed more dyadic operators than just <u>and</u> and <u>or</u>. Whereas my initial idea was an operator tentatively called DEFAULT, Jan Hajek finally succeeded in convincing me, after many long and heated discussions, that his idea, the ALT operator (described on page 76) was superior. It had more extra expressive power, it had symmetry and it had an extra error checking possibility. Hajek, whose emphasis was on <u>logic</u> and <u>operators</u>, later designed a whole new class of additional operators [Hajek, 1988]. These have not been included in SIMPLEXYS, where the emphasis was on <u>inferencing</u>.

Later, in December 1986, Hajek presented me with the source code of his expert system Quixpert, the logic of which was described in Hajek [1988]. Quixpert was extremely useful, too. The THELSEs (described on page 56) provided a breakthrough both in expressive power and in efficiency. And Quixpert's ASK rules (page 52) provides me with a last missing link: the concept of an <u>active</u> 'rule type'; Quixpert showed me how to evaluate 'primitive' rules only when necessary. I immediately invented other types of primitive rules as well, beside ASK rules. The first full-feature version of SIMPLEXYS, consisting of a Rule Compiler and an Inference Engine, was ready in March of 1987 and could do anything Quixpert could, at

least the Quixpert version of December 1986, and more, much more compactly, and possibly even faster. But then, that had not been the last version of Quixpert either...

From about May 1987 on, cooperation continued in a different way, at times officially sanctioned, at other times unofficially. Jan and I had fewer brainstorm sessions, but now he also started to take part in the coaching of the M.Sc. students who participated in the development of SIMPLEXYS. Boon [1987], whose task it was to develop the semantic checks, was the first. Hajek referred him to two fundamental building blocks: Quine's method (page 115) and the closure algorithm (page 118). In coaching him and others, Jan Hajek showed these students entries into the literature, and discussed problems, solutions and caveats. In his coaching, his contributions must have been substantial, too.

Jan Hajek's influence on SIMPLEXYS has therefore been pervasive. Only his most important contributions, the "micro expert" and Quixpert, have been mentioned in my dissertation, and not in very much detail. Many of Jan Hajek's contributions have not been mentioned because they were effectively invisible: providing entries into the literature, but most of all helping the students and me to hone and develop ideas, discard inferior ones and improve good ones. Jan, thank you again, for everything you have done. And in particular, thank you for your "destructive mind set", which has inspired many a good idea and saved me quite a lot of effort in pursuing bad ones.

J.A. Blom

STELLINGEN

behorende bij het proefschrift

The SIMPLEXYS Experiment

real time expert systems in patient monitoring

door

J.A. Blom

Eindhoven, 11 mei 1990

1. Natuurlijke taal kent, in tegenstelling tot een kunstmatige taal als SIMPLEXYS, geen 'primitieven'. Het begrip 'betekenis' is in een natuurlijke taal dan ook problematisch.

Dit proefschrift

2. Veel menselijke kennis is slecht onder woorden te brengen (het probleem van kennisacquisitie).

Dit proefschrift

3. Kennisacquisitie is de formalisering van intuïtieve overtuigingen.

B. Russell: De menselijke kennis
Servire, Katwijk, 1950

4. Een programmeertaal moet de communicatie mogelijk maken tussen een mens, die opdrachten geeft, en een machine, die de opdrachten uitvoert. De functie van de programmeertaal is deze communicatie zo foutloos mogelijk te laten verlopen.

L. Wittgenstein: Filosofische Onderzoekingen
Boom, Meppel, 1976

5. Definitieve criteria voor de geldigheid van uitspraken zijn voor het menselijk denken (ook voor het wetenschappelijk denken) niet weggelegd.

L. Kolakowski: Horror Metaphysicus
Kok Agora, Kampen, 1989

6. De betekenis van een begrip is niet altijd duidelijker naarmate we het vaker gebruiken.

P. Vroon: Intelligentie
Ambo, Baarn, 1980

7. De meeste van de bedoelingen die de programmeur had bij het schrijven van zijn computerprogramma gaan verloren bij het compileren van dat programma.

Dit proefschrift

8. Een gebeurtenis treedt op, zij treedt niet op, of het is onbekend of zij al dan niet optreedt (uitgangspunt van de SIMPLEXYS logica).

Dit proefschrift

9. Alle abstracties op hoog niveau behoren uiteindelijk te berusten op waarnemingsgegevens.

N. Elias: Een essay over tijd
Meulenhoff, Amsterdam, 1982

10. Waarnemen is zinloos als de waarneming niet kan leiden tot enigerlei aktie.

D.O. Hebb: Organizations of behavior;
a neurophysiological theory
Erlbaum, Hillsdale, 1963

11. Een waarneming kan slechts tot kennis leiden als bekend is hoe die waarneming geïnterpreteerd dient te worden. Het verkrijgen van kennis vooronderstelt dus kennis.

Dit proefschrift

12. Het achteloze gebruik van de uitdrukking 'niet te filmen' geeft er blijk van dat de spreker een geringe kennis bezit van de mogelijkheden van de hedendaagse cinematografie.