National Institute of Electricity and Electronics
INELEC - BOUMERDES
DEPARTMENT OF RESEARCH

THESIS

Presented in partial fulfilment of the requirements of the

DEGREE OF MAGISTER

In Electronic Systems Engineering

by

Mohamed AZNI

A Model Based Formal Specification of a Medical Expert System

Defended on july 12, 1994 before the jury:

President: Dr A.M. BOULARAS, Professeur, U.S.T.H.B
Members: Mr B. MEZHOUD, Maitre Assistant, INELEC
Dr H. AZZOUNE, Chargé de Cours, U.S.T.H.B
Dr M. DJEDDI, Chargé de Cours, INELEC

Registration Number 02/94

Dedications

To my mother

To my father

To my wife

To my sisters

To my brothers

ACKNOWLEDGEMENTS

I extend my most sincere gratitude to my research advisor Mr. Belkacem MEZHOUD for his continued interest and guidance throughout the present work.

I am also very grateful to Professor Mustapha MAAOUI, an expert surgeon at the Hospital of Thenia, for his availabilty and the precious time he accepted to sacrifice for us in order to offer the necessary expertise that is necessary for this project and without which this can never be achieved.

I would like also to thank Dr. K. HARRICHE for the useful criticisms and suggestions he made about this work, and for the encouragements and providence of the research facilities necessary to the success of this project.

My thanks are also due to Mr. Arezki BENFDILA for his moral support and encouragements.

Finally, I would like to thank all friends and INELEC staff who contributed to the achievement of this work.

ABSTRACT

In this thesis, the problem of the specification of the reasoning processes that are used in the development of expert systems is investigated, and a specification model of a medical expert system is proposed.

Traditionally, expert systems are developed by following an exploratory programming model of the software life-cycle. In this model of software development, the developer progresses towards the desired system by building an initial implementation, and, gradually, refining it into working prototypes, until a satisfactory system is obtained. The successive prototypes that are developed through the development process serve, actually, to explore the knowledge domain, and, hence, the user requirements of the system. In this model, there is, thus, no specification of the system before it is actually built. The reason for choosing such an approach for the development of expert systems is that the human reasoning processes are very difficult to specify. This makes expert systems subject to frequent changes.

Formal methods for software development are seen today as the most important achievement attained by the software engineering community in the development of new methods for software development. These mathematically based methods, when correctly used, allow the software developer to develop very high quality software systems in a cost-effective manner. However, a

fundamental step in using these methods is the writing of a precise and complete specification of the system requirements, before embarking in the costly subsequent steps of development. This makes them, somehow, unsuitable for the development of expert systems.

In this thesis, a method for using a model-based formal method, called Z, in order to produce a formal specification of a medical expert system for the diagnosis of the acute abdominal pain emergency is proposed. It is shown how, by integrating the Z-language into the formal system of fuzzy sets, it is possible to specify, using mathematical constructs, the approximate reasoning processes used by expert physicians during their diagnostic activities.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

In the past three decades, considerable improvements have been made in the microelectronics technology. With the advent of microprocessors in the beginning of the 1970s, increasingly powerful and cheaper computer hardware is being produced, and the size of the accompanying software systems is continuously increasing. With this availability of low cost and powerful computer hardware, computer systems are gradually proliferating, replacing mechanical or manual systems, into almost every area of activity in our modern life. Today, computers and their associated software are to be found in robots, motor cars, business administration systems, or safety critical systems such as aircraft control systems and nuclear power plants.

This proliferation of computers into the different activities raises, however, an important economical problem; whereas hardware production costs are low, the cost of developing the software that executes on that hardware is high, and it is expected that software costs will continue increasing for still some coming years [1]. The reason is simple; computer science is such a young discipline that we have not yet accumulated enough experience to permit us to develop large, high quality software

systems in a cost-effective manner.

The first programmable digital computers appeared during the 1940s. In those early days of computing (up the early 1960s) the problems solved using computers were quite well understood. They were, essentially, mathematically based problems, for which precise specifications could easily be formulated; for example, how to solve a differential equation. Programming at that time was then, essentially, the task of an individual. A physicist, for example, would develop his own program, written directly in machine language, to solve a particular problem in physics.

As computers became cheaper and more common, and with the introduction of high level programming languages, more and more people started using them. The size and the complexity of computer systems started to grow. Since most of computer systems are software intensive, the demand for larger and larger software systems started to be felt. With the increasing size of these software systems, software development could no more be handled by a single person; the applications that were proposed for automation have become so diverse and needed so large software systems that software development has rather become a team's activity. Within the team, each member would develop a small part of the complete system, and would integrate it, at the end, with parts from other team's members, to form the complete system. However, early large systems have seen the communication between team members to be rather poor.

In addition to the increasing size of software systems, there is the fact that, due to the diversity of the candidate applications for automation, the software developer and the end-

user have become different people. We cannot expect, for example, a business administration agent to develop his own software system. Such people, in general, have no background in computer science. This separation between the developer and the user, usually, raises the following problem: The developer, often, is called upon to develop software for application areas that he may not fully understand. This causes the developer to misinterpret the user requirements of the system. Misinterpretation of the user requirements leads, in turn, to poor user acceptance of the developed system.

Programming practice remained in the state described above up to the late 1960s. At that time, software development has arrived to an impasse. It has been discovered that the techniques of programming that were used in the early days for small programs could not be scaled up for large systems. The term "Software Crisis" has been invented at that time [1]. Software development was really in a crisis situation. Many projects were late and ran over budget. There was the need for better methods, tools, and techniques in order to develop better quality systems. New management techniques were also needed, in order to predict and control the development process.

The identification of the software crisis has led to the view that software development is an engineering discipline. The term "Software Engineering" was invented at the same time as the software crisis. The intent of software engineering is to define principles and to set guidelines in order to produce, within budget, better quality software systems.

It is worth noting that the problems of software development

and software engineering concern all kinds of software systems, including artificial intelligence programs. In this thesis, we intend to apply the principles of software engineering in order to develop a particular kind of artificial intelligence program, an expert system [2]. The application area that we have chosen is the medical domain.

Since the advent of computers, artificial intelligence has considerably evolved. Artificial intelligence tries to use the computer to create intelligent artifacts. A wide range of applications have been explored by the artificial intelligence community. Among the problems tackled by researchers in this field, we find: Natural language understanding, problem solving such as game playing and theorem proving, pattern recognition, and machine learning [3-5].

The decreasing cost of computer hardware was a catalyst for the emergence of highly specialized artificial intelligence programs known as expert systems [6]. These systems focus on a narrow domain and contain an impressive amount of expert knowledge in that particular domain. An expert system is a *complex* model of the decision making-processes of a human expert (psychologists use ***simple*** models, namely linear regression equations, for the same purpose. Complex versus simple models are compared in [7]). An expert system is intended to help non-specialists to solve significant problems in the particular area for which it was designed.

One of the application areas for expert systems that have been explored extensively is the medical domain. The motivation for the development of medical expert systems are the obvious

benefits that we can gain from a reliable and thorough diagnostic service. The task of a clinician confronted with an ill patient consists in gathering pertinent data and using it, in the light of his knowledge, in order to establish a diagnosis, and, eventually, recommend a treatment. In trying to find a diagnosis for a patient, the physician can omit some important data, and hence, miss the correct diagnosis. A computer program, however, can be designed to take into account all possibilities, thus increasing the chances to produce more reliable diagnoses. Furthermore, some tasks, such as calculating treatment dosages, can be done more rapidly and reliably by the computer than by the physician.

A lot of tasks in medicine have been identified as potential applications for the development of medical expert systems. Most of the systems developed have focused mainly on diagnostic systems. Diagnostic systems are, in general, designed to perform both the diagnosis task (finding the most likely disease or diseases) and the treatment recommendation task. Examples of representative systems in this category are MYCIN [8], CASNET [9], and CADIAG [10,11]. Outside the class of diagnostic systems, we can find, for example, monitoring systems [12], teaching systems, such as the PATHMASTER system [13], and systems for translation of medical documents, such as the TRANSOFT system [14].

In this thesis, we are concerned with expert systems that fall in the first category; that is, diagnostic systems. Our aim, here, is to develop a medical expert system to provide assistance for unexperienced practitioners in their decision-making

processes, in the area of the acute abdominal pain emergency. Experience in this area of medicine has shown that the acute abdominal pain emergency remains a difficult problem for unexperienced surgeons; the diagnostic accuracy of the surgeon depends largely on his past experience. This makes computer-aided decision making systems valuable tools in this area [15].

In order to apply the principles of software engineering for the development of our system, we need to rely on one of the standard methods for software development. The work in this thesis is centered around a formal method.

Formal methods for software development encompass a class of mathematically based methods that were designed for the development of large software systems. Formal methods provide a framework within which software developers can specify, develop, and verify, using mathematical concepts, software systems. Various formal methods have been developed, oriented each towards a class of applications. For example, for sequential systems, Z [16] and VDM [17] can be applied. For distributed and concurrent systems, CSP (Communicating Sequential Processes) [18] may be best suited.

A fundamental step in the development of a conventional software system, using a formal method, is to write a precise specification of the requirements on the system to be built. From the software engineering point of view, expert systems are developed in much the same way as conventional software systems. They are developed, installed, maintained, and eventually discarded. However, there is a fundamental difference between conventional software systems and expert systems [2]; expert

systems are knowledge-based systems. Since we do not yet understand well the nature of knowledge, it becomes difficult to specify the behaviour of an expert system, before actually building it. How, then, formal methods can be fitted in the development process of expert systems?

Traditionally, expert systems are developed by following an exploratory approach, that is, exploring the system requirements by building an initial implementation, and then, gradually, refine it until a satisfactory system is obtained. In this approach, thus, there is no specification. A recent experiment, however, by HILAL and SOLTAN [19], has been attempted in order to explore the specification approach and to compare it with the exploratory approach. The authors conclusion can be summarized as: The best approach to take is to combine both approaches with the depth of specification being dictated by the intended application.

In this thesis, a method of how the Z specification language can be used in order to develop a formal specification of a diagnostic system for the acute abdominal pain emergency is proposed. We shall see how medical knowledge can be captured, using mathematical constructs of the Z language, and how this can be used to make approximate inferences, required for the diagnostic task.

Our intention, through the present work, is to find a method in order to develop a medical expert system, in the light of the principles of software engineering, using one of the standard formal methods for software development. To approach the subject, the following structure is suggested:

In chapter 2, software engineering is defined first. Next, the qualities, characterizing any well engineered software system, are discussed. This is followed by a concise justification for the need of a precise specification of the requirements on a software system, before it is actually built. Having gained some experience with the principles underlying software development, the software process is, then, discussed. Two standard approaches, the structured approach and the formal transformations approach which are used in the process of software development are considered. Since we are concerned mainly by the latter, much emphasis is put on the formal approach. Chapter 2 is then terminated by mentioning software tools and management issues in software engineering.

Chapter 3 deals with the techniques of knowledge representation that are used in the development of expert systems. In fact, the problem of knowledge representation being the backbone of expert systems development, we have thought it necessary to include such a chapter to justify the choice, that is made later, of the technique which is used to represent medical knowledge in the system described in this thesis. In this chapter, focus is on the declarative scheme of knowledge representation which is the most widely used in practice.

Chapter 4 is the backbone of this thesis. It integrates both the principles of software engineering and expert systems by devising a method to produce a detailed specification of a model of a rule based medical expert system for the diagnosis of the acute abdominal pain.

Finally, there are two ingredients in this thesis which are

worth mentioning: the Z notation and the theory of fuzzy sets. Since our specification model is centered around the Z notation, we have thought it natural to explain how the notation can be used. Appendix A is included for this purpose. Fuzzy sets are used in this thesis in the purpose of handling approximate reasoning that is encountered in medicine. Thus, in appendix B a short review of this subject is given.

CHAPTER 2

AN OVERVIEW OF SOFTWARE ENGINEERING: Principles, Methods, and Tools

2.1 INTRODUCTION

The Term "Software Engineering" was invented in the late 1960s, when the software developers recognized for the first time the software crisis. At that time, there begun a heated debate about the process of software development. For the past three decades, researchers have been discussing the engineering principles that should underlie the process of software production, and they were striving for methods and tools that would permit a systematic development of software systems, in the light of the principles defined. Although real improvements have been achieved in our view and understanding of the process, the debate is still continuing today.

Software engineering has been studied extensively [1,20-22] and several comprehensive definitions for the subject have been formulated. As our working definition we take the following one

proposed by MARCO and BUXTON and reproduced from MARCO [20]:

Software Engineering is the establishment and use of sound engineering principles and good management practice, and the evolution of applicable tools and methods, and their use as appropriate, in order to obtain - within known but adequate resources limitations - software that is of high quality in an explicitly defined sense.

Although software engineering concerns all software systems, the benefits are most appreciable when the principles are applied to the development of large systems. The development of large systems comprises a variety of activities which require a wide range of skills. However, rarely a single person can have such a diversity of skills. Therefore, large systems are generally developed by teams of software engineers, rather than individual programmers, and, hence, software engineering becomes, then, the problem of coordinating the activities of the different members of the team in order to develop well engineered products in a cost-effective manner.

2.2 SOFTWARE QUALITIES

It is clear from the above definition that the intention of software engineering is the development of high quality products. It is then important that software qualities be explicitly defined. A well engineered software system is characterized by the following qualities:

♦ ***Maintainability***: Software systems are dynamic in nature, they evolve over their life-time. Maintenance refers to the activity of making modifications to the system, that arise, due to changes in a variety of system requirements; for example, correcting errors that are discovered after delivery of the system, adapting the system to a new environment, or enhancing the system and adding new features to it. The experience gained by software engineers in the development of some large projects has shown that the maintenance phase absorbs about 60% of the total system costs [1]. Therefore, the system must be built in such a way that it anticipates change, thereby, easing maintenance and reducing total system costs.

♦ ***Reliability***: An unreliable system is of no use. Reliability means that the system must deliver the intended services for which it was developed. That is, it must behave according to its specification.

♦ ***Efficiency***: The system must be efficient. This means that the system should make an economical use of resources, such as space and time.

♦ ***User interface***: The system must present an adequate interface to the user. Adequacy means that the system must be easy to use. Adequacy being a subjective quality, it will, in general, depend on the expected user of the system.

It must be noted that the above qualities are not mutually

independent; one can affect the other. For example, we can maximize the efficiency quality but, at the expense of reducing the maintainability quality, since increasing efficiency will, in general, increase the complexity of the system. Depending upon the intended system's application, a trade-off is generally attained between these software qualities. For example, in the case of medical expert systems, SHORTLIFFE [23] identified the reliability and the user interface criteria to be among the most important attributes to consider, if the systems are to be accepted by practicing clinicians in hospitals.

2.3 THE IMPORTANCE OF SPECIFICATION

In the previous section, we have seen that the maintenance phase absorbs the majority of the total system costs. However, a deep thought has led software developers to the conclusion that the majority of the errors that are discovered during the maintenance phase are errors of omission, which are actually traceable to the definition phase. This, in turn, has led to the view that, if more interest is put on the definition phase, then major changes in the maintenance phase could be avoided, thereby, reducing appreciably the total system costs. The importance of writing a precise and complete specification of the system requirements, before embarking in the costly development process, has then become evident. In fact, requirements definition and specification is so important that deep research efforts have been done for developing methods and techniques for systems analysis, and entire books have been written on the subject [24].

2.4 THE SOFTWARE PROCESS

The software engineering discipline provides a framework within which software systems are developed. The process of actually applying the principles of software engineering to create and maintain real products is called the software production process. This process extends over time, and goes through a number of steps, with each step defining explicitly the activities which are carried out. If the steps are ordered in some structure then this structure will define a software life-cycle model or a software engineering paradigm.

At the most abstract level, the process of software production can be seen to be broadly divided into three generic phases which are:

- ***Definition***: During the definition phase, the problem for which an automated system is desired is analyzed as completely as possible. The expected system services are explicitly defined. Functional and non-functional requirements as well as the desired performances and design constraints are precisely defined.

- ***Development***: The development phase focuses on how solutions to the problem defined can be provided using an automated system.

- ***Maintenance***: In the maintenance phase, we take care of the eventual changes that are necessary to bring about to the system. Changes can appeal to corrective, adaptive, or enhancement maintenance.

2.5 SOFTWARE ENGINEERING PARADIGMS

We have now defined the principles underlying the process of software development and a generic view of the phases of the process. We shall, now, consider some elaborate models of software development. We shall begin with the models used with the structured approach of software development. There are many such models, however, only some representatives in the structured approach are considered below. The formal transformations approach is considered later, in this chapter.

2.5.1 The Structured Approach

In the structured approach, each of the stages through which the process of software development goes, is explicitly shown. The stages are distinct. The order of the stages and the relationships between them define a structure.

2.5.1.1 The Waterfall Model

The waterfall model is the most widely used model of software development because it is very attractive from the management point of view. This model is shown in Figure 2.1. Its constituting phases are as follow:

♦ *System feasibility*: During this phase, an overall estimate of the system resources requirements is made. The resources are identified and a decision is made of whether or not the system can be built given the existing hardware and software technologies, and budget. The output of this phase is a

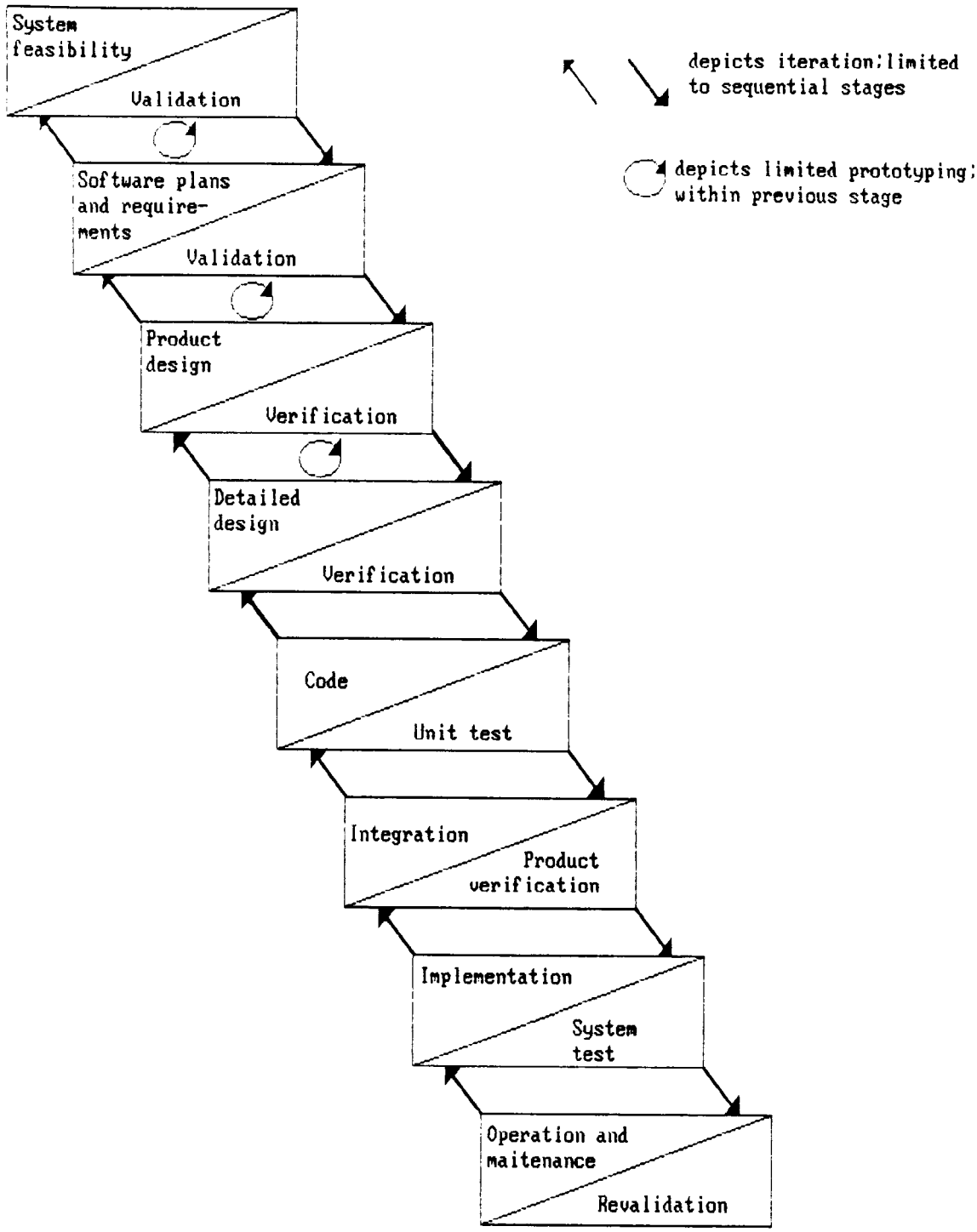


Figure 2.1: THE WATERFALL MODEL. [28] p. 41.

feasibility study report. The feasibility study is then validated. The activity of validation consists in evaluating the response to the question: "Are we building the right product ?"; in other words, we try to make sure that the intended system services are understood and captured fully and correctly.

♦ *Software plans and requirements*: During this phase, the user needs and desires, as well as the system functions and constraints, are defined. This phase is conducted with both the system developer and procurer participating together. The output of this phase is normally a requirements specification document that will act as a basis for a contract between the developer and the client. The requirements are, once again, validated.

♦ *Product design*: Based on the specification produced in the previous step, a logical solution is sought. The system is subdivided into subfunctions that can readily be transformed into program modules. The design is then *verified*. The verification activity evaluates the answer to the following question: "Are we building the product right ?"; in other words, we check that the design is a correct solution for the problem at hand.

♦ *Code*: The program modules are translated into computer programs, using some executable programming language. Each program module is tested for proper functioning.

♦ *Integration*: The program modules are assembled and integrated into a complete system. A verification activity of the integrated

system is carried out.

♦ *Implementation*: The complete system is installed and complete testing is conducted.

♦ *Operation and maintenance*: The system is released and installed. Once the system is operational, the maintenance phase normally starts, which consists in correcting errors that have slipped to the designer through the previous steps and, modifying the system because of a new requirement that might later arise.

2.5.1.2 Prototyping.

The model for the prototyping approach is shown in Figure 2.2. This approach is useful when the system user cannot define precisely his needs, or when the developer is unsure about the best tools that are appropriate to the current problem. The idea behind this model is to built quickly a working prototype system which is presented to the user for evaluation. The user can then identify new requirements and the developer can gain further insights into the problem. The developed prototype should, thus, be seen as a means of assessing and refining the user requirements.

The process of prototype development might need to be done more than once but, eventually the customer and the developer agree on a working prototype. Prototypes, however, lack the required software qualities. Thus, the next task is to develop a well engineered product and deliver it.

The criticisms that can be done about the prototyping approach are the following:

- The client, when seeing a working prototype, may be impatient and require that the prototype be delivered prematurely, ignoring that the prototype lacks the required qualities.
- During the prototype development, the developer may take inappropriate design decisions and implementation compromises. With time, he might forget, leaving those compromises which, obviously, decrease the system quality.

2.5.1.3 Fourth Generation Techniques.

This model is depicted in Figure 2.3. In this model, the idea is to use a very high programming language (called fourth generation languages) to generate executable code automatically from the requirements specification. Fourth generation languages are software tools that enable the software developer to specify some characteristic of software at high level, then generate automatically source code based on the developer's specification.

Like the other models, it begins with the requirements specification phase followed by the design phase. The design is transformed into a working program using a fourth generation language. Of course, there is some feedback from the later stages back to the earlier stages, as required. The final task is that of testing; the developer must perform complete testing and debugging of the system and produce relevant documentation, to end up with a deliverable product.

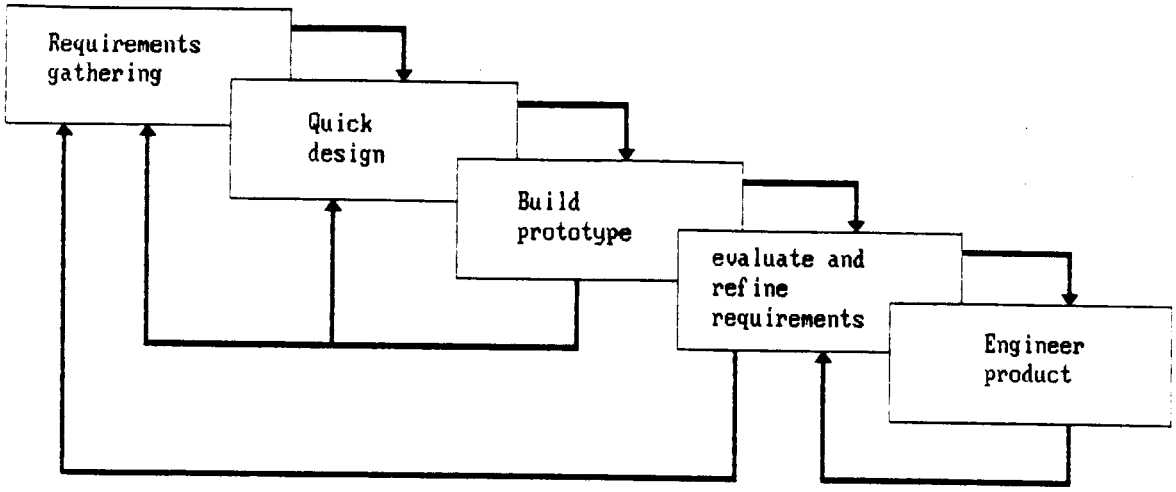


Figure 2.2: PROTOTYPING. [21] p. 23

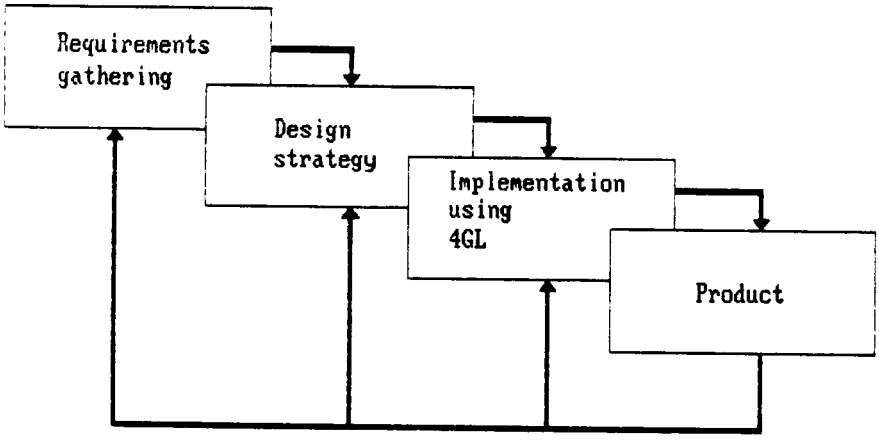


Figure 2.3: FOURTH GENERATION TECHNIQUE [21] p. 24

2.5.1.4 Exploratory Programming.

See Figure 2.4. This model resembles the prototyping model. However, there is one important difference: whereas in the prototyping model there is a requirements specification phase, in the exploratory approach we only develop an outline specification. Thus, in this latter model, there is no verification activity since the verification activity involves checking an executable program against its specification. The exploratory programming approach is mainly used in the development of artificial intelligence programs, where computer programs intend to emulate the reasoning processes of humans. In this field, it is generally difficult to specify the requirements of the system before actually building it.

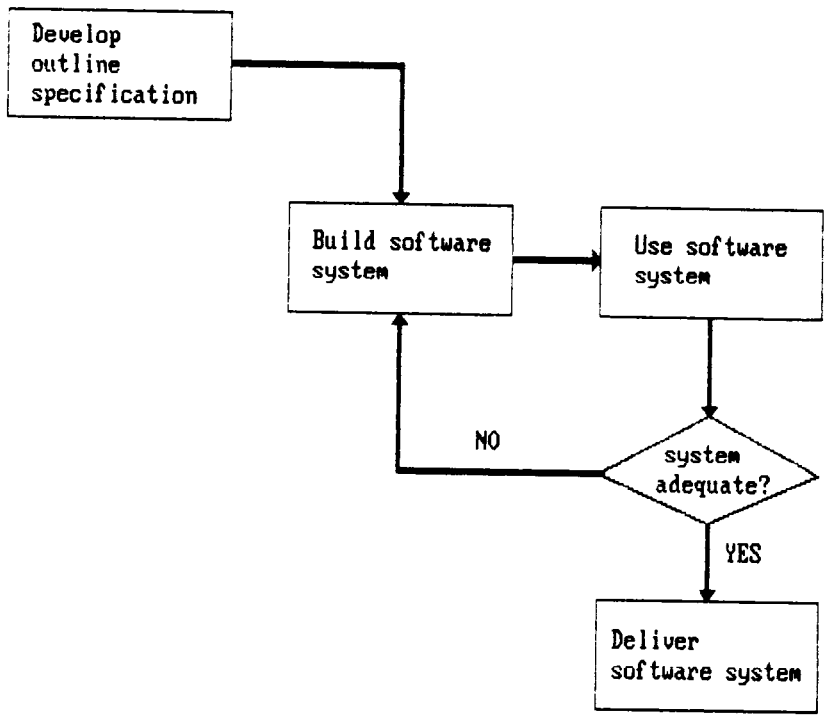


Figure 2.4: EXPLORATORY PROGRAMMING. [1] p. 12.

2.5.2 Structured Methods for Software Development.

We have now defined the principles of software engineering and we have discussed some models of software development. However, to drive software development, we need methods and tools to support the application of those principles. We shall now review some methods of analysis and design which are, generally, used in the structured approach.

The Methods which are generally used in the structured approach can be classified under the term *structured methods.* Structured methods rely on graphical techniques to carry out the software process stages in a Structured Analysis/Structured Design (SA/SD) approach [25]. Structured methods use a number of graphical techniques for systems analysis and systems design, such as Data Flow Diagrams (DFDs), Entity Relationship Diagrams (ERDs), and Entity Life History (ELHs), to describe different views and aspects of a system. For example, in the Structured Systems Analysis and Design Methodology (SSADM) [26], DFDs are used to describe the process view, and ELHs are used to describe the timing view. Other examples of structured methods are the YOURDON method [27], and the JSD method (Jackson Systems Development) [28].

2.5.3 Formal Methods for Software Development.

The use of formal methods for software development requires a software development life cycle which is fundamentally different from those used in the structured approach, discussed previously. With formal methods, the development is based on a series of formal transformations. The formal transformations

step will be the source specification for the next development step. The development activity is the process of transforming a source specification into a target specification. The verification activity is the process by which we can verify, ***by formally proving***, that the target specification conforms to the source specification.

From the model, it is clear that a prerequisite to enter the development process is the availability of a formal specification for the first development step. In other words, we need an initial source specification to start the process. It is obtained through the analysis and specification of the collection of user requirements, as shown in Figure 2.6. In Figure 2.6, it is clearly shown that this activity involves the participation of both the system developer and system procurer, and what is shown as precise specification constitutes usually the basis for a contract that is signed by both participants.

2.5.3.2 Application of Formal Methods in the Process of Software Development.

At the level of each development step in the model of formal transformations, the activities of development and verification are conducted using the so-called ***formal methods***. Formal methods encompass a class of ***formal systems***, that were specifically designed to provide the software engineer with a framework within which he/she can write precise specifications of large and complex systems, reason about them, and check for their satisfiability and consistency. (Formal systems that underlie the background necessary for the understanding and use of formal

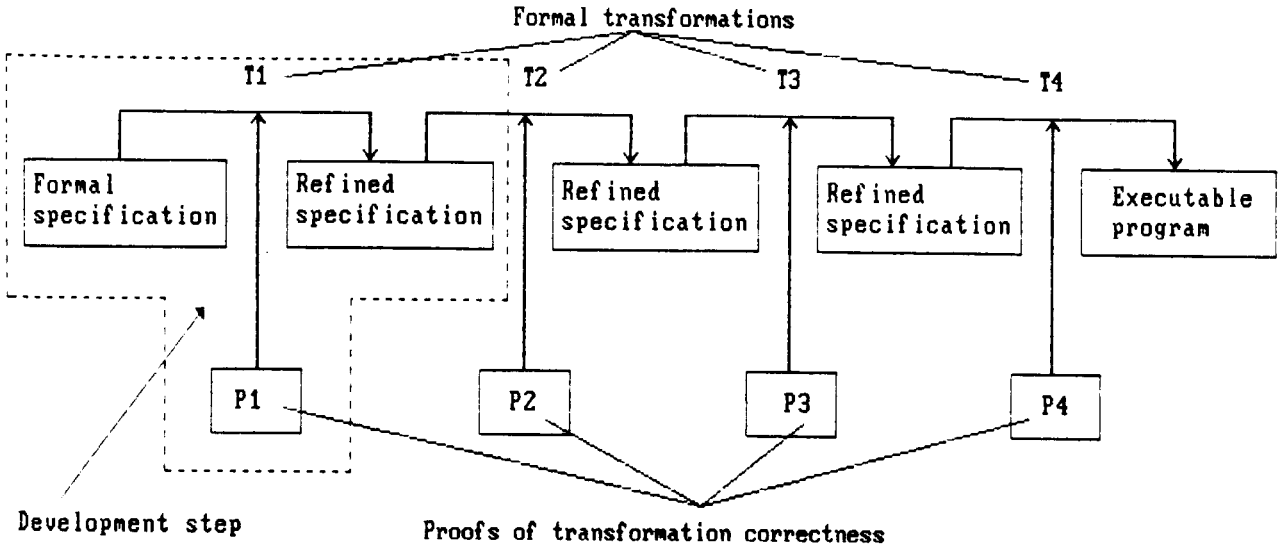


Figure 2.5: THE FORMAL TRANSFORMATIONS APPROACH. Adapted from [1] p.127 and [35].

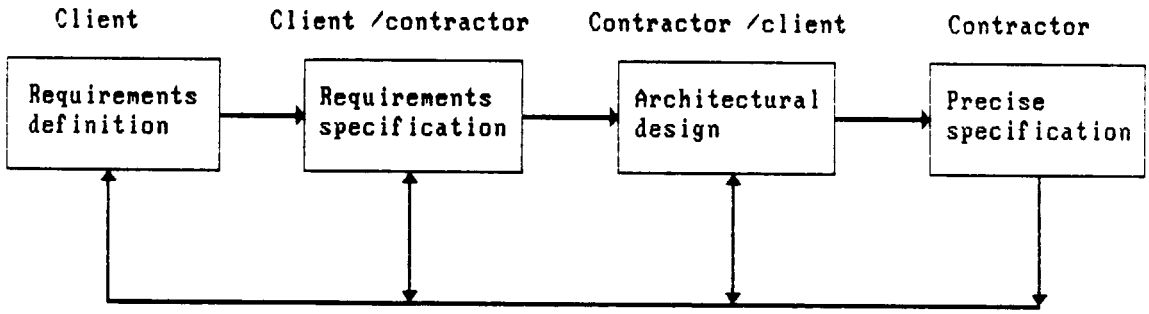


Figure 2.6: FORMAL SPECIFICATION DEVELOPMENT [1] p. 125

methods for software engineering are discussed in detail in WOODCOCK [29]). These methods rely on a sound mathematical base, and they make extensive use of mathematical structures such as sets, relations, functions, and sequences.

Since formal methods are essentially formal systems, we expect them to possess both a formal language and a deductive system. The formal language on which a formal method relies to write statements about the behaviour of a software system is called the ***specification language*** of the method.

In the past few years, various formal methods have been developed to handle different kinds of tasks arising in software development. These methods can be classified according to the semantical basis of their languages. Two classes of formal methods can be distinguished: Model-based methods and property-based methods.

Model-based methods (also called state-based) serve mainly to describe sequential systems. They allow the developer to build an abstract mathematical model of the state space description of the system to be developed. The development process progresses, then, by gradually refining the model into more concrete models, until an executable program is obtained. Methods that fall in this category include, for example, Z [16] and VDM [17]. An example of a model-based method that is dedicated to the description of concurrent and distributed systems is CSP (Communicating Sequential Processes) [18].

The property-based methods are further divided into two sub-categories: axiomatic methods, which are based on first-order predicate logic, and algebraic methods, which are based on multi-

sorted algebras. An example of axiomatic methods is Anna [30] and an example of the algebraic methods is Clear [31].

Another aspect to consider regarding formal methods concerns the executability of the languages on which they are centered. Most of the specification languages that have been developed thus far are not executable. The argument that is usually used against executability is that it restricts the expressiveness of the language. On the other hand, executability can be very useful, because it can permit a rapid development of prototypes thereby allowing early validation of the system. In a recent paper, FUCHS [32] has made an important claim. He showed that nonexecutable specifications can be made executable by rewriting them in a Logic Specification Language, and keeping a high level of abstraction.

2.5.3.3 Advantages and Disadvantages of Formal Methods

Formal methods have many advantages compared to structured methods. They provide the software developer with powerful abstraction facilities. This is due to the richness and the high degree of abstraction of the mathematical data types that they integrate in their notations. Abstraction offers some structuring mechanisms and, consequently, some complexity control. In fact, abstraction is an important issue in the specification of a software system. Abstraction allows the developer to focus on ***what*** the system is expected to do, without constraining him to take into account the implementation details.

Another advantage of formal methods is that the semantics of each statement in a specification written in the formal

language of a formal system can be unambiguously defined. Thus, the software engineer can check formally for the consistency of the specification.

A third advantage is due to the deductive system of a formal method. The deductive system contains axioms and inference rules. Thus, it becomes possible to reason about specifications. Specification being collections of mathematical entities, one can prove, using mathematical techniques, properties about the specification which help discover possible omissions and inconsistencies sufficiently early in the project. Furthermore, given a formal programming language and a formal specification, it can be possible to prove that an implementation conforms to its specification (verification activity), and then the testing phase becomes unnecessary. In the case where complete formal verification cannot be possible, formal methods can, at least, help identify a set of tests to apply on the system.

Some practical experience about the application of formal methods in industry and the benefits gained already exist. For example, the Software Engineering Project, the Distributed Computing Project, and the Transaction Processing Project have all been specified using the Z specification language [33]. In the case of the Transaction Processing Project, IBM has reported an overall reduction of their system (Customer Information Control System (CICS)) cost of 9%. In the area of safety-critical systems, BOWEN and STAVRIDOU [34] have summarized the software applications that have been developed using formal methods. Again, the authors noted that some appreciable benefits have been reported.

Although formal methods have many advantages as we discussed above, they still do not find wide applications in industrial scale projects. The reasons are numerous. Some of the reasons are due to the nature of formal methods themselves, some other are due to management. we summarize these reasons below.

♦ *education*: Current practitioners in software development are not technically trained for the use of formal methods. The application of formal methods requires some background in discrete mathematics and logic. To overcome this problem, researchers believe now that universities should include this background in their curricula.

♦ *Tools*: Much of the research in formal methods has been directed towards the development of notations and techniques to apply them. The development of tools that can assist the developer has rather been poor. However, tools are necessary for the effective use of formal methods. Type checkers and proof assistants are very desirable, especially when the specifications are large.

♦ *Management*: Managers are usually reluctant to formal methods because the benefits that they can gain from their adoption is not obvious for them. In other words the role of formal methods in the software industry is not clear for them. Indeed, it is not clear even for the proponents of formal methods. Nevertheless, researchers in formal methods are aware of this and some research efforts are being done in this direction. For example, PLAT et

al. [35] have discussed in detail the application of VDM in a standard model for software development, the DoD-STD-2167A and POLACK [36] has attempted to integrate Z with SSADM.

2.6 SOFTWARE TOOLS

Software tools play an important role in the process of software development, particularly for large systems. There are quite enough tools supporting the structured methods, both during the analysis and the design phases. However, they are relatively expensive. A survey of such tools can be found in [37] with details of one of them, the EXCELERATOR. In our institution (INELEC), a tool, AUTO-MATE PLUS, that supports the Logical Structured Data Modelling (LSDM) method is available. This tool is from the Learmonth and Burchet Management Systems (LBMS) company.

For formal methods, the current availability of software tools is low [38]. Although formal methods are starting to break through in the software industry, there is still much work to do before they become mature enough.

2.7 MANAGEMENT ISSUES IN SOFTWARE ENGINEERING

The software process models which we have discussed above are very useful for those who actually develop software systems. However, they are also important for manager of software projects. They allow some insights into the different activities and hence some management control on the development process.

Management issues are as important as the techniques of software development. Management issues include tasks such as project estimation, project scheduling, task decomposition and assignment. Management issues in software engineering are currently an active area of research (see, for example, [39]).

CHAPTER 3

KNOWLEDGE REPRESENTATION

3.1 INTRODUCTION

An expert system is a knowledge-based system which is designed to reproduce as closely as possible the reasoning processes of a human expert to solve significant problems in a particular area of expertise. An expert system comprises essentially a knowledge base, an inference engine, and some support software interfacing to the environment. A typical structure of an expert system is shown in Figure 3.1.

The knowledge base consists of an explicit representation of the knowledge which we think the human expert makes use of to solve his problems. This knowledge is domain specific and it is usually a collection of rules and facts to apply.

The inference engine is the main part of the system. As its name implies, it uses the knowledge, contained in the system's knowledge base, to draw appropriate inferences to achieve some desired goal. The inference process is done in steps, with each step creating additional knowledge. The inference engine is also able, through its interface, to get, at times, the information

it needs and to deliver its response to the user.

The support software, or interface to the environment, consists essentially of two parts. The first part interfaces with the user. Through it, the system can respond to the user, ask for further information, and explain its reasoned response. The second part is optional. It interfaces with the expert who constructs the knowledge. Through it, the expert can update, modify, or enrich the knowledge base.

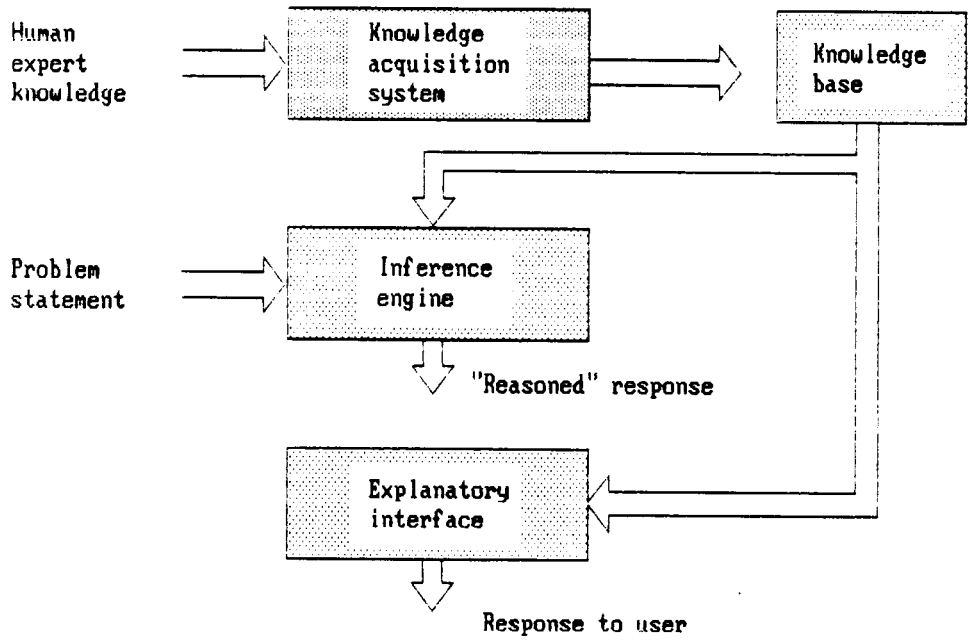


Figure 3.1: TYPICAL STRUCTURE OF AN EXPERT SYSTEM

3.2 KNOWLEDGE REPRESENTATION

In the process of development of an expert system, we are immediately faced with the problem of knowledge elicitation. That is, collecting (by interviewing an expert) all the relevant knowledge that we incorporate in the knowledge base. This knowledge is of two kinds: domain knowledge which includes rules, facts, and heuristics and control knowledge which applies the domain knowledge to create additional knowledge needed to solve the problem at hand. Having collected the appropriate knowledge, we are, next, faced with the problem of knowledge representation. This consists in deciding about a form of representation suitable for computer processing.

The problem of knowledge representation has a crucial role in the development of expert systems. In fact, knowledge representation being a formalism supporting the studied phenomena, the inference power of the subsequent system depends largely on the chosen representation. In a typical system, both domain knowledge and the control knowledge have to be modeled and represented in some form.

There are two fundamental forms of knowledge representation: procedural representations and declarative representations [3]. In procedural representations, explicit relations exist between the knowledge elements. These relations are established by the control mechanism. The control mechanism is a set of procedures which define exactly what knowledge elements should be used in a given situation and how. In declarative representations, the domain knowledge is presented as a collection of independent

pieces of knowledge, leaving to a control structure the task of manipulating these pieces and making deductions.

In this thesis, we will not consider further the procedural representation scheme. We shall only make a brief comparison of the two techniques. The main advantage of the systems having a procedural representation is their efficiency. However, as the size of the knowledge base grows, these systems become hard to understand and modify. On the other hand, systems centered around the declarative representation formalism have the advantages that, since the domain knowledge is independent from the control mechanism, modifications can be easily done, and the reasoning steps closely followed. However, compared to the systems using the procedural representation formalism, these systems are inefficient.

In the remaining of this chapter, we will review the main techniques developed by researchers in the declarative formalism.

3.3 DECLARATIVE REPRESENTATION

In the declarative representation scheme, we can address the problem in two fundamental approaches. In the first approach, knowledge is represented in clausal form. A set of well defined rules of inference permit a syntactic manipulation of these clauses to make mechanized deductions. This is the logic based knowledge representation. In the second approach, knowledge is generally expressed in the form of rules of implication. It is this second approach, which is the most widely used in real systems, that we shall consider more deeply in the remaining of

this chapter.

3.3.1 Production Systems

Production systems were originally developed by Post (1943). In this formalism, expert knowledge is represented in the form of a large number of simple production rules which direct the dialogue between the system and the user, to deduce conclusions. A production system consists of three parts:

- ***A set of production rules, or the rule base.***

The general expression of a production rule is
IF < conditions > THEN < actions >.
Where the conditions and actions parts are given in the form of assertions. A rule can be activated, or fired, if its conditions part, or antecedents, is satisfied. That is, if the conditions part exists as facts, known to be true, in the data base. If a rule is fired then its consequents, or actions, are deduced and added to the data base.

- ***A data base.***

The data base is the work area of the system. It is a short term memory that consists of facts which are known to be true about the particular problem to solve. The data base is also called the context.

- ***A rule interpreter.***

The rule interpreter applies the knowledge available in the knowledge base and, by reference to the data base, deduces a

given goal. At any time during execution, the task of the interpreter is to find the list of satisfiable rules, to select one of them, and to fire the selected one and add the deduced fact to the data base. The interpreter searches for the applicable rules by a process of pattern matching. That is, by matching the antecedents of each rule against the contents of the data base. If more than one rule are found applicable then a conflict arises. The problem, then, is to determine which rule to fire first. Practical systems, in general, provide some means of conflicts resolution.

Reasoning Directions.

There are two fundamental ways of deductive inference in production systems. The first approach is data driven, called *forward chaining.* The second approach is goal driven, called *backward chaining.* Let us illustrate these two approaches with the following small example.

Suppose that the knowledge base contains the following three rules:

R_1: IF A AND B THEN C

R_2: IF B AND C THEN D

R_3: IF A AND D THEN E,

and that the data base contains the known facts A and B.

Suppose that the goal of the system is to demonstrate the truth of E.

Forward Chaining

A forward chainer would start from the given facts A and B,

apply R_1 to demonstrate C, then apply R_2 to demonstrate D, and finally it would apply R_3 to demonstrate E which is the desired goal.

Backward Chaining

A backward chainer, however, would start from the goal E and proceed backwards. In an attempt to demonstrate E, the interpreter would apply R_3 to generate two sub-goals A and D. A being verified, it remains to verify D. The interpreter would then apply R_2, generating the sub-goals B and C. Finally, The application of R_1 would demonstrate the truth of C because A and B are verified (given). Thus the sub-goal D would be established which, in turn, establishes, with A, the goal E.

Control Structures in Production Systems

In a typical production system, it is often the case that more than one production rule are applicable at a particular time. A conflict between the applicable rules, then, results. The set of conflicting rules is called the ***conflict set***. To resolve the conflict, the interpreter must be able to choose which rule to apply first. There are generally three methods for conflict resolution.

1) Exhaustive application of the relevant rules. This is an elementary method which does not take into account the evolution of the database.
2) The rules are activated according to some criterion. For example, the highest priority rule is applied first, where the level of priority is defined by the programmer according to

the demands and characteristics of the task.

3) Use of meta-rules to reason on the object rules. Meta-rules are similar, in form, to the object rules. They include heuristics and operate on the object rules.

3.3.1.1 Example of a Production System: MYCIN [4,8]

The MYCIN system was developed by SHORTLIFFE et al. at the University of Stanford (U.S.A). It was designed to provide nonspecialist physicians with consultative advice on the diagnosis and treatment of infectious diseases. The name MYCIN comes after the common suffix for several antibiotics. The structure of this system can be seen in Figure 3.2.

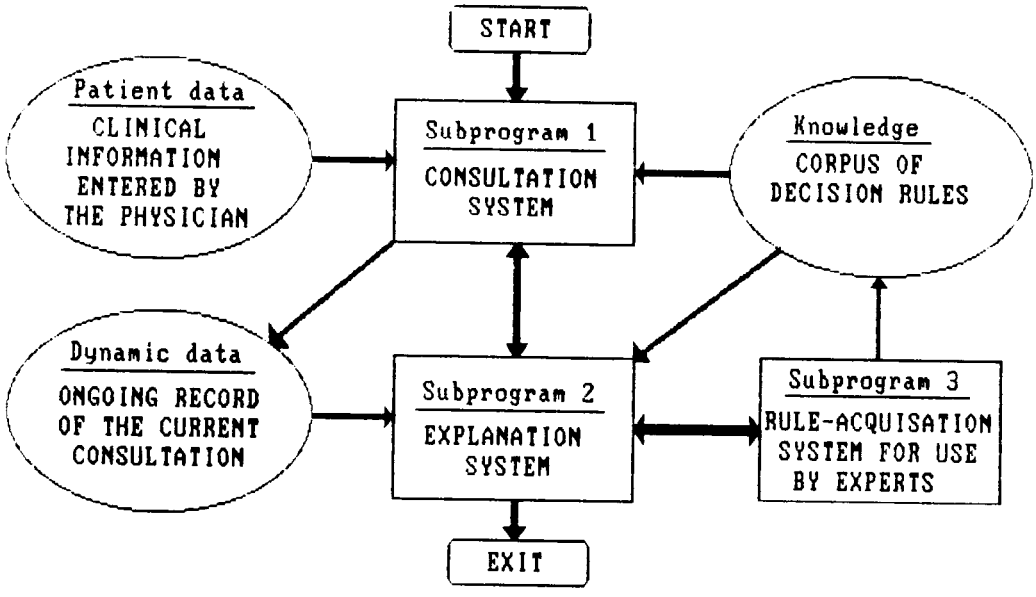


Figure 3.2: STRUCTURE OF THE MYCIN SYSTEM [8].

As shown in Figure 3.2, MYCIN consists of three interacting sub-programs that form the essential system. Each of the sub-programs handles a sub-task of the complete consultation task.

The consultation system is responsible for: determining whether or not the infection necessitates a treatment, identifying the organism(s) responsible for the infection, selecting a list of drugs administrable to the current patient, and choosing from this list one or more drugs to prescribe.

The explanation system permits the consulting physician to ask '*why* questions' if the system requests suspicious information to the user, and '*how* questions' if the system comes up with a questionable conclusion. The explanation system allows also the expert who built the knowledge base to follow the reasoning steps of the system. In that way, if the expert does not agree with a certain system's conclusion, he can always identify the source for the disagreement.

The rule acquisition system can acquire directly from the expert new rules or modify existing rules, thereby continuously enhancing the performances of the system.

MYCIN's Knowledge Base

The knowledge base of MYCIN consists of about 200 therapeutic decision rules that are coded internally in LISP. Each rule consists of a premise part and a conclusion part. The rules are weighted with Certainty Factors (CF) in the sense that, if the premise part of a given rule is satisfied then its conclusion part is deduced, with a degree of confidence equal to the weighting CF. CFs are numbers ranging between -1 and +1,

where -1 represents complete confidence that the conclusion is false and +1 represents complete confidence that the conclusion is true. An example of such a rule is:

IF: 1) THE STAIN OF THE ORGANISM IS GRAMNEG, AND
    2) THE MORPHOLOGY OF THE ORGANISM IS ROD, AND
    3) THE AEROBICITY OF THE ORGANISM IS ANAEROBIC
THEN: THERE IS SUGGESTIVE EVIDENCE (.6) THAT THE IDENTITY
      OF THE ORGANISM IS BACTEROIDES.

The premise part of each rule is always a conjunction of simple clauses. The clauses are represented as 4-tuples in the following form:

<PREDICATE> <OBJECT> <ATTRIBUTE> <VALUE>.

For example, in the first clause of the rule above the predicate function is <IS>, the object is <ORGANISM>, the attribute is <STAIN>, and the value is <GRAMNEG>. There are some 24 predicate functions in MYCIN that are evaluated to true or false depending upon the value of the CF of the attribute.

Reasoning

In MYCIN, the rules are invoked in a backward chaining fashion. The relevant rules to the current goal (or sub-goal) are considered exhaustively. This results in a depth-first search of an AND/OR goal tree. The CF of a goal (sub-goal) is calculated according to the following:

♦ For each rule the CF of the resulting conclusion is the product of the CF weighting the rule and the minimum of the CFs of the clauses constituting the premise of that rule.

♦ If a conclusion is attained by two different rules then the CF of the conclusion will be given by:

$$CF = CF_1 + CF_2 - CF_1 * CF_2,$$

where CF_1 and CF_2 are the CFs of the two rules.

The final important aspect of MYCIN, to consider, is its control structure. The control structure of MYCIN consists of meta-rules. An example of such a meta-rule is the following:

IF: 1) THE INFECTION IS A PELVIC ABSCESS, AND
    2) THERE ARE RULES WHICH MENTION IN THEIR PREMISE ENTEROBACTERIACEAE, AND
    3) THERE ARE RULES WHICH MENTION IN THEIR PREMISE GRAM-POS-RODS,
THEN: THERE IS SUGGESTIVE EVIDENCE (.4) THAT THE FORMER SHOULD BE DONE BEFORE THE LATTER.

For each sub-goal, these meta-rules (heuristics) are examined, thereby reducing or reordering the set of rules to be used.

3.3.2 Semantic Networks

A semantic network is a directed graph, consisting of nodes and directed arcs, or links, interconnecting those nodes. The nodes represent objects, concepts, or situations. The directed links represent the relationships that exist between the nodes. As an example of semantic networks, consider the level of pathophysiological states in the CASNET system [9] (see Figure 3.3). In this level of the system, the nodes represent the states through which a disease evolves and the links represent causal relationships between the disease states.

The semantic network representation has the following interesting features:

- The relationships between objects are always binary relations. This makes it very easy to add new knowledge to the knowledge base.

- All the important facts that one wants to say about objects in a given domain are explicitly represented. Thus, relevant facts about an object can be inferred directly from the nodes to which they are linked without a search through a large database.

A semantic network is a natural way to represent declarative knowledge about relationships, such as the membership relation, between objects in a given domain. This can be very useful if, for example, a classification of the objects in that domain is desired.

Inferences in a semantic network are coded in procedures

that manipulate the network structure. However, it is clear that there are already some elements of control, in the data structures coding the information, that guide the search.

3.3.2.1 Example of a Causal Network Based System: CASNET [9].

The Causal ASsociational NETwork (CASNET) is a medical expert system developed by WEISS et al., at the University of Rutgers (U.S.A), for the diagnosis and treatment of glaucomas. In this system, medical knowledge is represented in a particular type of sematic network consisting of three levels of knowledge.

The three level knowledge representation is shown in Figure 3.3. As shown in Figure 3.3, each level corresponds to a description of one aspect of the disease processes. The level of observations contains the evidence, including signs, symptoms, and test results, about the patient. The level of pathophysiological states in a causal network of states describing the evolution of the diseases. In this level, the nodes represent elementary hypotheses about the disease process and the links highlight the causal relationships that exist between them. The third level contains the classification tables for the disease. These tables define a disease category as a set of confirmed and denied pathophysiological states.

The links in the causal network of states are weighted with numbers, ranging between 0 and 1, indicating the strength of causation. These numbers are interpreted in terms of frequency of occurrence. Starting states are assigned a starting frequency. The relation between the level of pathophysiological states and the level of observations is associational. Associational links

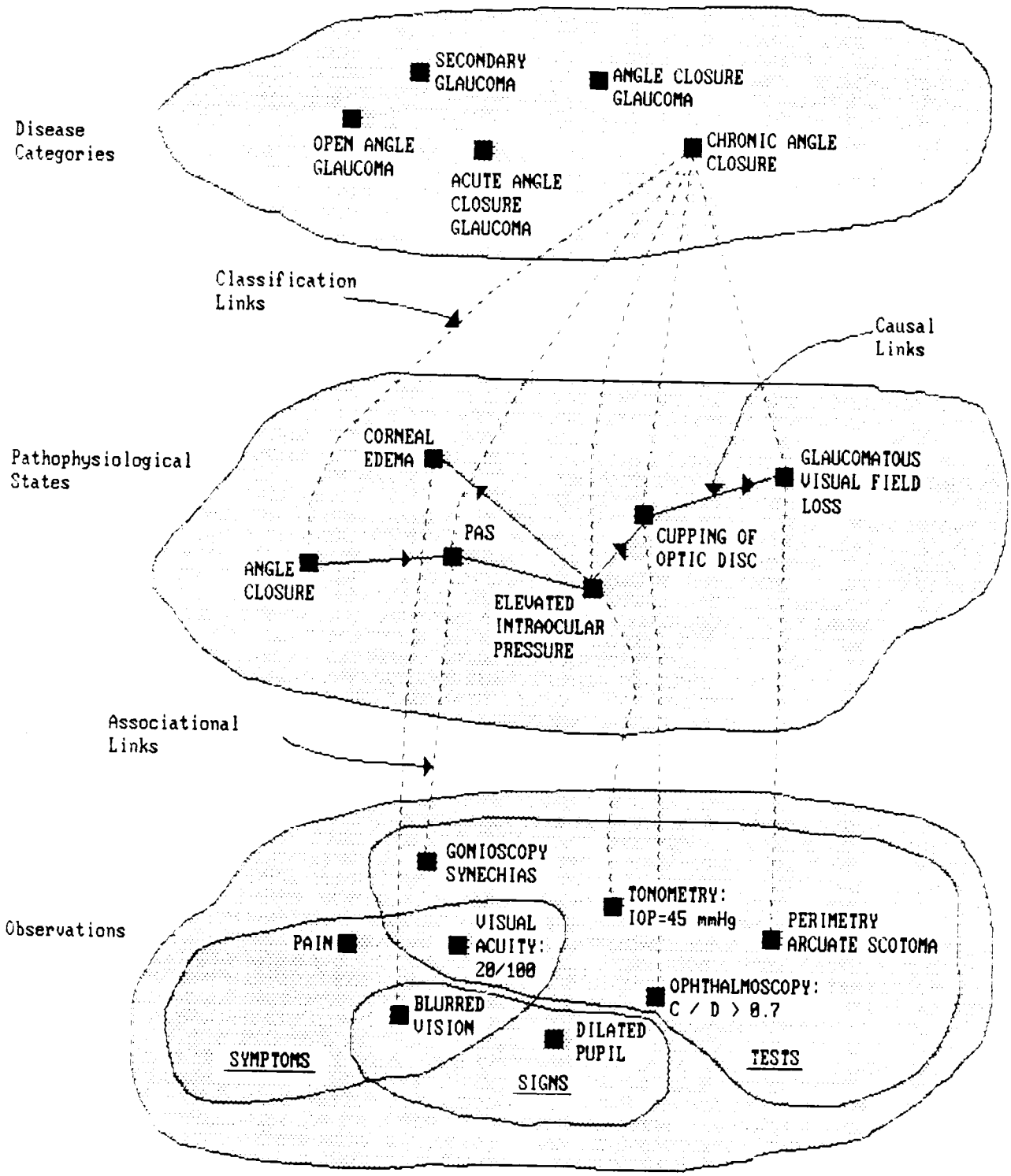


Figure 3.3: THREE LEVEL DISEASE DESCRIPTION IN CASNET [9].

between these two levels are also weighted with numbers ranging between -1 and +1.

Reasoning

The rules of inference in the CASNET system are as follows: Let $Cf(n_j)$ be the confidence factor associated with the node (disease state) n_j, t_i be any test or observation, and Q_{ij} be the weight associated with the associational link between t_i and n_j. Initially, the Cf of all nodes is undetermined ($Cf(n_j)=0$).

Rule 1: When a test result is received and a rule associating t_i with n_j is found applicable, then

(a) If $|Cf(n_j)| < |Q_{ij}|$ then $Cf(n_j)$ is reset to Q_{ij}.

(b) IF $Cf(n_j)= -Q_{ij}$ then $Cf(n_j)$ is set to 0 until another result t_k is received such that $|Q_{kj}| > |Q_{ij}|$.

(c) Otherwise $Cf(n_j)$ is unchanged.

Rule 2: Let α be a threshold number between 0 and 1.

(a) If $Cf(n_j) > \alpha$, then n_j is assumed confirmed.

(b) If $Cf(n_j) < -\alpha$, then n_j is assumed denied.

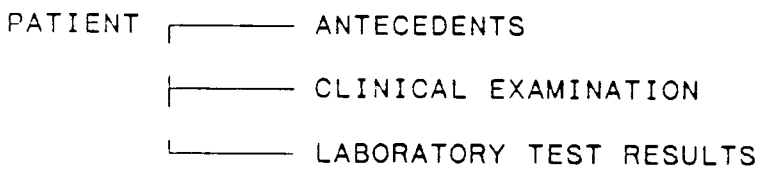
(c) Otherwise, the status of n_j is assumed undetermined.

3.3.3 Frames

A frame is a data structure consisting of a mixture of declarative and procedural knowledge that represent a typical situation (stereotype) in a domain of the real world. The idea behind the frame representation is that human memory stores typical structures of information, and that we evoke one of these

structures each time we are in a given situation, trying to match that structure to the given situation.

A frame contains slots to store the facts that are typically known about the situation that the frame intends to represent. Slots can themselves be other frames, or they can be simple identifiers. The frames describing all the aspects of a given situation are related between them, and they usually form a tree. An example of a frame may be the description of a typical patient. A frame describing a typical patient can have slots, for example, for: antecedents, clinical examination, and laboratory test results as shown below:



If we fill in the slots of a frame with a particular kind of information then we say that we have created an instance of the frame. Thus, in the patient frame above, if, for example, for a particular patient we find that he matches a structure describing a typical patient suffering from cancer, then we conclude that our patient is, in fact, suffering from cancer.

The search in frame-based systems is guided by procedures of inference that are attached to each slot. The procedures are of two kind. The first kind concerns the knowledge needed to control the search for obtaining the information needed to fill in the slots. In the absence of relevant information slots can be filled with default values. The second kind are procedures

that explicit the actions to be taken after a given slot has been filled. For example, deciding that the current situation matches the selected frame, or transferring control to another frame if a match does not occur.

3.3.3.1 Example of a Frame Based System: INTERNIST [4].

The INTERNIST system was developed at the University of Pittsburgh (U.S.A), by H. Pople and J. Myers. It was designed for the diagnosis of diseases in the area of internal medicine.

Knowledge about diseases in the INTERNIST system is organized into a disease tree, as shown in Figure 3.4.

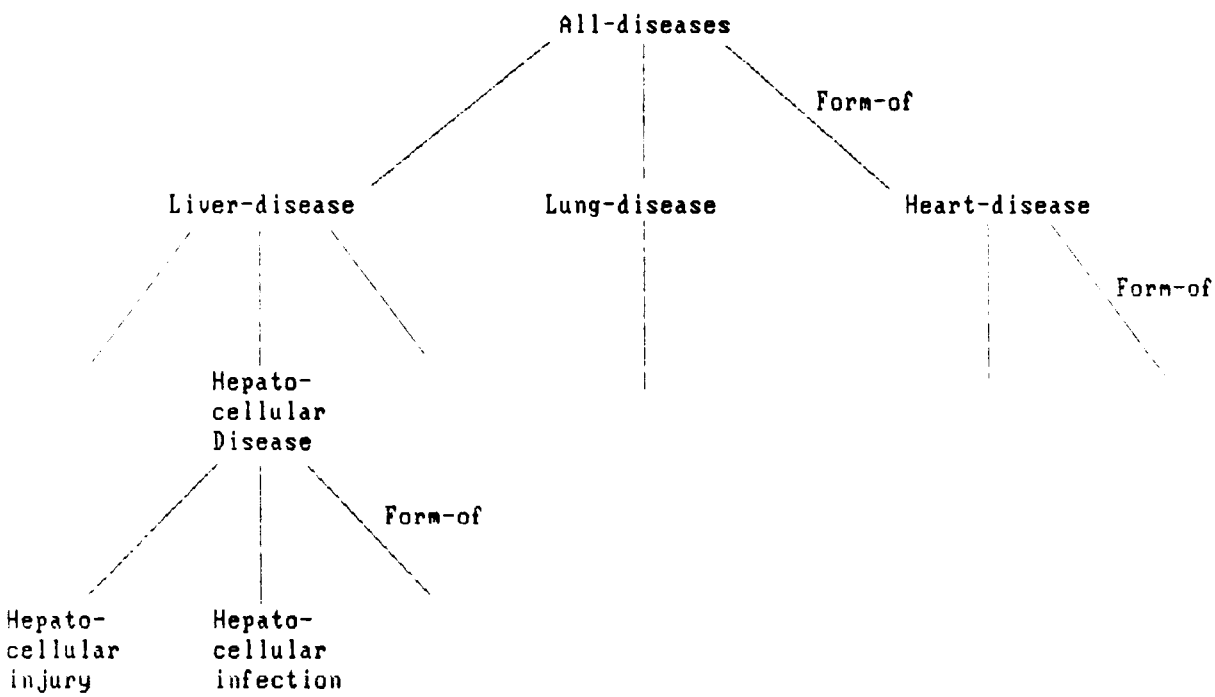


Figure 3.4: DISEASE TREE IN INTERNIST [4] p. 198.

In this tree, the nodes refer to diseases. A terminal node is called a ***disease entity***. A non-terminal node and its sub-tree is called a ***disease area***.

Each disease entity can be seen as a frame two slots of which are used for two types of relations, relating disease entities to manifestations. These relations are: the EVOKE relation (a manifestation can *evoke* a disease) and the MANIFEST relation (a disease can *manifest* a disease). The relations EVOKE and MANIFEST are weighted with real numbers between 0 and 5 indicating the strength of relationship between diseases and manifestations.

Manifestations are associated with various properties, among which, the most important are the TYPE and the IMPORT properties. The TYPE of a manifestation indicates the cost of asking to test for that manifestation. Thus, the less expensive tests are asked for first. The IMPORT property indicates how much important a manifestation is for a given consultation.

Each frame for a disease entity or a disease area contains also a slot for the relation *form of* relating that disease entity or disease area to the other disease entities or areas in the tree. This relation permits disease entities to inherit some of the properties associated with the nodes in its sub-tree. Thus, for example, a manifestation can evoke a disease area if the manifestation evokes all of the nodes in its sub-tree.

Reasoning

Each of the manifestations entered to the system during a consultation session evokes one or more nodes of the disease

tree. INTERNIST, then, creates for each evoked node a *model* consisting of four lists:

- A list of observed manifestations, called the *shelf*, that this disease cannot explain;
- A list of observed manifestations that are consistent with the disease;
- A list of unobserved manifestations that should be present in the patient if this is the correct diagnosis;
- A list of manifestations not yet observed but which are consistent with the disease.

A diagnosis in the INTERNIST system corresponds to a set of evoked terminal nodes that explains all manifestations, taking into account their IMPORT property. To establish a diagnosis, the system tries to reduce as much as possible the set of such terminal nodes if that set is large, or tries to discriminate between them if that set is relatively small. Based on the models described above, the system selects from the appropriate lists the appropriate questions.

We have now summarized the main techniques of knowledge representation that have been used to build expert systems, and have given some examples of systems that have been built, using these techniques. In general, these systems have shown good performances. However, researchers in artificial intelligence are claiming that still more efforts need to be done to discover new techniques to permit more performant systems [40].

CHAPTER 4

A MODEL-BASED FORMAL SPECIFICATION OF A DIAGNOSTIC SYSTEM FOR THE DIAGNOSIS OF THE ACUTE ABDOMINAL PAIN.

4.1 INTRODUCTION

The primary objective that was set to achieve through the work in this thesis is the development of a *formal specification* of a medical expert system. The specification should be centered around one of the well known formal methods, namely, the Z method. However, the term "medical expert system" being vague, it has become necessary to restrict ourselves to a particular area of medicine. This restriction is necessary for two reasons:

1) The choice of a particular knowledge representation technique, to represent medical knowledge, is, in general, dictated by the nature and the particularities of the application area.
2) Medical practice can be best appreciated only by questioning experts about the way they tackle the problems with which they are daily faced.

The development of an expert system involves the

participation of an expert. We have, thus, developed the specification that follows after about ten interview sessions, of two hours each, with an expert in the area of the acute abdominal pain. The developed specification is concerned only with the diagnostic phase of the consultation process. The motivation for building such a model system is twofold:

♦ The diagnosis of the acute abdominal pain emergency is a difficult problem for unexperienced surgeons, and experience with computer-based systems has shown that such systems can be very helpful for those young surgeons [15].

♦ The need to write precise requirements specification for software systems, before they are actually built, being today recognized, and due to particularities of expert systems as compared to conventional systems [2], it is interesting to investigate if there is a way for a formal method, such as Z, to be used for the development of such systems.

The problem that we tackle in this thesis is, thus: to use the Z specification language in order to specify a model for an expert system that will provide medical-decision making assistance, for unexperienced surgeons, in the diagnosis of the acute abdominal pain. Indeed, we have already developed such a model [41], and in the remaining of this chapter, we shall review in more details how the model was obtained.

4.2 INTRODUCTORY REMARKS

We shall start by discussing the problem of how to handle

uncertainty, which is inherent in medical decision making, in computer-based systems performing approximate reasoning.

The problem of handling uncertainty in systems performing medical reasoning has evolved over the best part of the last three decades. Early systems were based on statistical methods, and they focused on the diagnosis phase of the consultation process. They assumed that complete data is available "en block", and they performed statistical calculations to infer the most likely diagnosis, based on the entered data and past experience [42,43]. However, it was rapidly discovered that the method was not practical, because it is not in all cases that complete data is available, and when the data becomes large, the method results in a combinatorial explosion.

The problems related to statistical methods have motivated researchers to investigate in the sequential decision-making [44] and the knowledge-based paradigm. These systems are able to reason with incomplete information, and they handle uncertainty in a manner different from that of statistical methods. For example, the MYCIN [8] and CASNET systems [9] use confidence factors to weigh uncertainty, and they propagate evidence using simple formulae. For a discussion on the relative merits of statistical and knowledge-based systems, see [45]. Other systems, such as the CADIAG system [10,11], and [12] rely on fuzzy set theory [46] to perform approximate reasoning (see appendix B for a short review of fuzzy sets).

In recent years, new techniques, based on causal probabilistic networks (CPNs) have been developed. CPNs are seen, today, as a new kind of knowledge representation technique. A CPN

is a causal network of states and weighted links [47-49]. The nodes represent medical entities and the links represent causal relations with their weights indicating *a priori* probabilities of the strength of causation between the nodes. CPNs originated from the work by KIM and PEARL [50]. CPNs are today evaluated very positively [51]

In this thesis, we have chosen to use fuzzy set theory as a means to handle uncertainty. Our choice of fuzzy set theory is not guided by some kind of superiority of fuzzy sets over the other techniques. Instead, we have found it suitable to integrate the Z notation in it and, thus, to model the approximate reasoning processes of the expert.

4.3 THE Z-MODEL OF THE SYSTEM

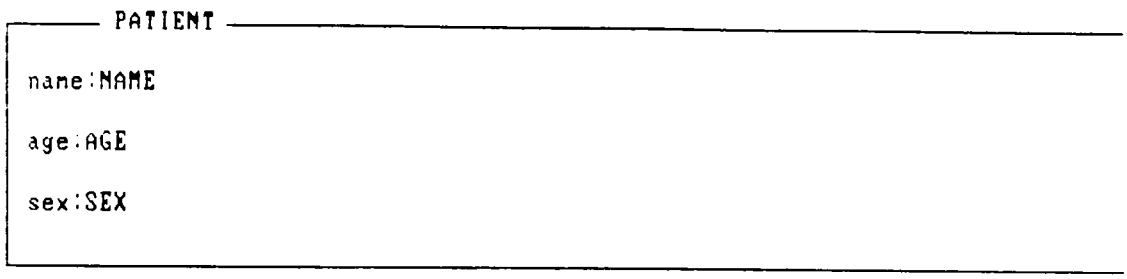
4.3.1 General Considerations

In the language of Z, a specification of a software system is normally presented in a bottom-up style; that is, one starts by defining the sets that are primitive to the specification (primitive data types). In this thesis, we follow that style of presentation.

Let us denote by PATIENT the set of all patients that can present to a hospital for an examination. A particular patient is identified by his name, sex, and age. These elements are drawn, respectively, from the sets NAME, SEX, and AGE.

[NAME, SEX, AGE].

We use a schema to identify a patient by his name, sex, and age.



A patient can suffer from a certain ***disease*** or a ***combination of diseases***. An ill patient will, in general, show a class of symptoms. In this thesis, we refer by ***symptom*** to any of the subjective sensations such as nausea and chest pain, or the objective signs and laboratory findings which are observable by the physician. In the area of interest to us (acute abdominal pain emergency), the expert physician, based on his experience, can enumerate the diseases and symptoms that characterize an acute abdominal pain. Thus, we consider the sets DISEASE and SYMPTOM to be primitive to our specification.

[SYMPTOM, DISEASE].

Medical reasoning is always subject to some degree of uncertainty. Symptoms and diseases are usually weighted (***linguistically***), by clinicians, to indicate the degree of belief that they have in a patient showing a given symptom, or suffering from a certain disease. For computer processing, it is helpful to associate the linguistic weights (such as ***always***, ***never***, ***almost always***, and ***almost never***), used by clinicians, with real

fractional numbers. Based on the theory of fuzzy sets, the method to calculate the real numbers that are assigned to the linguistic weights, such as those mentioned above, is explained in detail in [10]. In the specification that follows, knowledge about symptoms and diseases is weighted with real number between 0 and 1. We specify, thus, the set FRACTION of real numbers between 0 and 1 using a schema; (by the way, the following is a schema in horizontal form);

FRACTION = [r : R | 0 =< r =< 1].

Since we will make use of fuzzy sets, fuzzy relations and their compositions, we need to specify formally the membership function characterizing these mathematical structures. Let X, Y, and Z be three generic sets (parameters) of arbitrary objects. If a fuzzy relation, relating objects from X to objects from Y, is characterized by a membership function *f*, then *f* can be specified formally, in Z, by the following axiomatic definition:

$$f : X \times Y \rightarrow\!\!\!\!\!+ \; \mathbf{FRACTION}$$

and the membership function, that we denote by *1-f*, characterizing the complement of the fuzzy relation characterized by *f*, may be formally specified, in Z, by the following generic constant:

$$[X, Y]$$

$$f : X \times Y \pfun FRACTION$$

$$1\text{-}f : X \times Y \pfun FRACTION$$

$$dom(1\text{-}f) = dom\ f$$

$$\forall (x,y) \in dom\ f \bullet (1\text{-}f)(x,y) = 1\text{-}f(x,y)$$

If we are given two fuzzy relations characterized, respectively, by the membership functions *f* and *g* then a Z specification of their composition may be written as:

$$[X, Y, Z]$$

$$\_ \circ \_ : (X \times Y \pfun FRACTION) \times (Y \times Z \pfun FRACTION) \pfun (X \times Z \pfun FRACTION)$$

$$\forall f : X \times Y \pfun FRACTION \bullet \forall g : Y \times Z \pfun FRACTION \bullet$$

$$f \circ g = \lambda f \bullet \lambda g \bullet$$

$$\lambda x : X \bullet \lambda z : Z \mid (x, y) \in dom\ f \wedge (y, z) \in dom\ g \bullet y \in Y \bullet$$

$$MAX \{ r : FRACTION \mid r = MIN \{ f(x,y), g(x,z) \} \}.$$

4.3.2 Knowledge Representation

In medicine, the specification language Z can be best applied in areas where it is possible to obtain a descriptive model of the disease mechanisms that are involved, because the deeper the level of knowledge that we have about the disease processes, the better appear the relationships between the different processes that take place in an organism, and the easier these relationships can be captured using the mathematical

structures of the Z language. For example, the program developed by TODD [52] for the diagnosis of nerve lesions in the peripheral nervous system has shown impressive performances.

The diagnosis of the acute abdominal pain emergency remains a difficult problem, and this usually requires expert knowledge. Diagnostic accuracy in this field is generally low. The problem is even worse for unexperienced surgeons. The reason for this difficulty seems to be related to the insufficient amount of medical knowledge that has been gathered in this field, by clinicians. In fact, it appears that the disease mechanisms involved in the acute abdominal pain emergency are, unfortunately, not well understood. Experts in this field seem to rely more on their past experience than on their understanding of the processes of diseases evolution. Their reasoning is rather judgemental, and they infer their conclusions simply by establishing associational relationships between observed symptoms and diseases. The following example shows the kind of medical knowledge that drives the diagnostic process of clinicians in this area.

EXAMPLE

"La CHOLECYSTITE AIGUE atteint les hommes mais surtout les femmes. L'histoire du malade est charactérisée par des douleurs à l'hypochondre droit avec des irradiations à l'épaule droite. A l'éxamen physique, le malade réagit au test de Murphy, a une vesicule palpable avec des douleurs à l'hypochondre droit, et a une temperature moderée. Un tel patient nécéssite une opération urgente ou semi-urgente".

In areas where most of the reasoning is based on subjective judgements, production rules can be the best suited technique of knowledge representation. In the case of the diagnosis of the acute abdominal pain, production rules are, in our opinion, suitable to represent medical knowledge. In this thesis, we have, thus, represented medical knowledge in the form of production rules. We consider two kinds of rules relating symptoms to symptoms, diseases to diseases, and symptoms to diseases. The first kind of rules is related to the concept of ***occurrence***. The general form of such rules is, for example:

- IF the disease *d* is found in a patient, THEN we expect that the symptom *s* occurs with the disease *d* WITH degree of occurence belief (certainty) *r*.

The second kind of rules is related to the concept of ***confirmation***. The general form of such rules is, for example:

- IF the symptom *s* is shown by a patient, THEN the presence of disease *d* in the patient is confirmed, WITH degree of confirmation belief *r*.

Occurrence and confirmation are seen, here, as fuzzy relations relating objects from the primitive sets SYMPTOM and DISEASE (see [11]). Each kind of rules gives rise to three fuzzy relations. In the following, we specify, in Z, the characteristic functions of the resulting six fuzzy relations;

R^{o}_{SD} : SYMPTOM × DISEASE ⇸ FRACTION

R^{c}_{SD} : SYMPTOM × DISEASE ⇸ FRACTION

R^{o}_{SS} : SYMPTOM × SYMPTOM ⇸ FRACTION

R^{c}_{SS} : SYMPTOM × SYMPTOM ⇸ FRACTION

R^{o}_{DD} : DISEASE × DISEASE ⇸ FRACTION

R^{c}_{DD} : DISEASE × DISEASE ⇸ FRACTION

The result returned by each of the functions specified above is a measure of the degree of membership of a couple of medical variables (symptoms or diseases) to either of the fuzzy relations *occurrence* or *confirmation*. For example, the function R^{o}_{S} indicates how often a symptom occurs with another symptom, and the function R^{c}_{DD} indicates how often a disease confirms another disease when that disease is found in a patient. The functions are, of course, documented by the expert surgeon, and they form the knowledge base of the system which are specifying. They are partial, because there might be variables (symptoms or diseases) for which the expert surgeon has no judgement concerning the relationships between them.

4.3.3 The Consultation Process

The specification of the consultation process that follows is based on three medical rules of inference, used by ADLASSNIG et. al. in the development of their system, CADIAG [10,11]. These three rules of inference are: the symptom-to-disease inference, symptom-to-symptom inference, and the disease-to-disease inference. In the sequel, these inference rules will be stated at the appropriate levels of the specification.

The consultation process starts by the activity of recording the pattern of symptoms shown by the patient under examination. Each of the recorded symptoms is assigned, by the consulting physician, a degree of belief, indicating the confidence he has in the patient showing that symptom. Weighting the recorded symptoms is necessary, because the physician may be uncertain that the patient under examination shows a given symptom. We shall call MANIFESTATION the set of symptoms recorded by the consulting physician. We specify the state space description of the pattern of symptoms recorded, by a schema:

PatientSymptomPattern

$$patient : PATIENT$$

$$MANIFESTATION : \mathbb{P}\ SYMPTOM$$

$$PatientDescription : PATIENT \times SYMPTOM \pfun FRACTION$$

$$dom\ PatientDescription = \{ patient \} \times MANIFESTATION$$

In the schema given above, the function *PatientDescription* can be interpreted as being the characteristic function of a fuzzy relation ***shows*** (a patient ***shows*** a symptom). The function is gradually built, as the symptoms are recorded. The domain of the function is restricted to the symptoms recorded, i.e., MANIFESTATION, and that concern the patient under examination.

Now, we define one operation on the pattern of recorded symptoms; the operation is that of adding a symptom to the already recorded list of symptoms. The operation is called *AddRecordedSymptom*, and we specify it using a schema, as follows:

AddRecordedSymptom

ΔPatientSymptomPattern

symptom? : SYMPTOM ; r? : FRACTION

---

patient' = patient

symptom? $\notin$ MANIFESTATION

MANIFESTATION' = MANIFESTATION $\cup$ { symptom? }

PatientDescription' = PatientDescription $\cup$ { (patient, symptom?) $\longmapsto$ r? }

The operation *AddRecordedSymptom* is shown to change the state of the pattern of symptoms. There are two inputs to the system: the physician enters the newly recorded symptom and the weight that he assigns to it. As a precondition to the operation, the symptom to record must not be in MANIFESTATION, i.e., it must not be already recorded. If the precondition is satisfied, then the operation has the effect of augmenting the set MANIFESTATION

and the function *PatientDescription*, to take the new symptom into account.

4.3.3.1 Symptoms-to-Diseases Inferences

We shall now see how the inferences from symptoms to diseases are performed. We will consider three rules of symptoms-to-diseases inference compositions which are: the symptom-to-disease confirmation composition (we call it $COMPOSITION_1$), the symptom-to-disease positive exclusion composition (we call it $COMPOSITION_2$), and the symptom-to-disease negative exclusion composition (we call it $COMPOSITION_3$).

Rule 1. Symptom to disease confirmation composition.
($COMPOSITION_1$)

- IF a symptom ***symptom*** is believed to be shown by the patient ***patient*** under examination, with degree of belief r_1, i.e., *PatientDescription(patient,symptom)* = r_1, AND
  the system "knows" that whenever a patient shows the symptom ***symptom*** **(with degree of belief = 1)**, then the patient suffers from the disease ***disease*** with degree of ***confirmation*** belief r_2, i.e., $R^{c}_{SD}(symptom,disease) = r_2$, THEN
  it can be inferred that the patient ***patient*** is suffering from the disease ***disease*** with degree of belief r_3 such that:

$$r_3 = \text{MIN}\ [r_1, r_2].$$

- If more than one symptom confirm the same disease with

different degrees of belief, then the symptom with the greatest degree of confirmation belief is taken to perform the inference composition. This inference rule is applied to infer for all diseases for which it is meaningful.

The specification of the above rule of inference is given in the schema below:

COMPOSITION$_1$

PatientSymptomPattern

KNOWNMANIFESTATION : $\mathbb{P}$ MANIFESTATION

KNOWNDISEASE : $\mathbb{P}$ DISEASE

R^1_{PD} : PATIENT $\times$ DISEASE $\rightarrowtail\!\!\!\!\!\!\rightarrow$ FRACTION

KNOWNMANIFESTATION $\times$ KNOWNDISEASE $\subset$ dom R^c_{SD}

R^1_{PD} = PatientDescription $\circ$ ((KNOWNMANIFESTATION $\times$ KNOWNDISEASE) $\lhd$ R^c_{SD})

In the schema above we have introduced two sets: KNOWNMANIFESTATION and KNOWNDISEASE. The set KNOWNMANIFESTATION is the set of symptoms recorded, for which the inference rule stated above is meaningful. In other words, the set KNOWNMANIFESTATIONxDISEASE must be in the domain of R^c_{SD}. Also, given the set KNOWNMANIFESTATION, the system can apply the inference rule above only for diseases for which it "has knowledge" of symptom to disease confirmation relationships. Thus, the inference rule is valid only for the set KNOWNDISEASE of diseases such that SYMPTOMxKNOWNDISEASE is included in the

domain of $R^{\hat{}}_{SD}$. The constraint in the first line of the predicate part of the schema COMPOSITION$_1$ says, in fact, that the inference rule is valid only for the intersection of KNOWNMANIFESTATIONxDISEASE and SYMPTOMxKNOWNDISEASE.

Rule 2. Symptom to disease positive exclusion composition.

(COMPOSITION$_2$)

- IF a patient ***patient*** shows a symptom ***symptom*** with degree of belief r_1, i.e., *PatientDescription(patient,symptom)* = r_1, AND the system "knows" that whenever a patient shows the symptom ***symptom*** **(with degree of belief = 1)**, then the patient suffers from the disease ***disease*** with degree of ***confirmation*** belief r_2, i.e, $R^{\hat{}}_{SD}$(*symptom,disease*) = r_2 ,THEN
  it can be inferred that the patient patient is ***not*** suffering from the disease *disease* with degree of belief r_3 such that:

$$r_3 = \text{MIN}\ [r_1, 1-r_2].$$

- If more than one symptom confirm the same disease, then the symptom with the greatest degree of confirmation belief is taken to perform the inference composition. The inference rule is applied to infer for all diseases for which it is meaningful.

The specification of the above rule of inference is given in the following schema:

$COMPOSITION_2$

PatientSymptomPattern

KNOWNMANIFESTATION : $\mathbb{P}$ MANIFESTATION

KNOWNDISEASE : $\mathbb{P}$ DISEASE

R^2_{PD} : PATIENT $\times$ DISEASE $\pfun$ FRACTION

KNOWNMANIFESTATION $\times$ KNOWNDISEASE $\subset$ dom R^c_{SD}

R^2_{PD} = PatientDescription $\circ$

(KNOWNMANIFESTATION $\times$ KNOWNDISEASE) $\lhd$ $(1-R^c_{SD})$

Rule 3. Symptom to diseases negative exclusion composition ($COMPOSITION_3$).

- IF a symptom ***symptom*** is believed to be shown by the patient under examination with degree of belief r_1, i.e., *PatientDescription(patient,symptom)* = r_1, AND
  the system "knows" that whenever a patient suffers from a disease ***disease*** **(with degree of belief = 1)**, then the symptom ***symptom*** occurs with ***disease***, with ***occurrence*** degree of belief r_2, i.e., R^0_{SD}(*symptom,disease*) = r_2, THEN
  it can be inferred that the patient under examination is ***not*** suffering from the disease ***disease*** with degree of belief r_3, such that:

$$r_3 = \text{MIN}\ [1-r_1, r_2].$$

- If it is known that more than one symptom occur with the same disease, then the symptom with the greatest degree of

occurrence belief is taken to perform this inference composition. This inference is applied to infer for all diseases for which it is meaningful.

The specification of this rule of inference is given below:

$COMPOSITION_3$

PatientSymptomPattern

KNOWNMANIFESTATION : $\mathbb{P}$ MANIFESTATION

KNOWNDISEASE : $\mathbb{P}$ DISEASE

R^3_{PD} : PATIENT $\times$ DISEASE $\pfun$ FRACTION

KNOWNMANIFESTATION $\times$ KNOWNDISEASE $\subseteq$ dom R^o_{SD}

R^3_{PD} = (1-PatientDescription) $\circ$

((KNOWNMANIFESTATION $\times$ KNOWNDISEASE) $\lhd$ R^o_{SD})

4.3.3.2 Symptom-to-Symptom Inferences

In medicine, it appears that certain symptoms ***always occur*** with some symptoms and ***never occur*** with some others. It appears also that the presence of certain symptoms in a patient ***always confirm*** the presence of some symptoms and ***never confirm*** the presence of some others. Experts in medicine are aware of this, and they take that into account, during their diagnostic activities. In the model which we are describing, we take this into account. As for the symptom-to-disease inference, we shall specify the symptom-to-symptom confirmation inference composition, the symptom-to-symptom positive exclusion inference composition, and the symptom-to-symptom negative exclusion

inference composition.

Rule 4. Symptom to symptom confirmation composition.
(COMPOSITION$_4$).

- IF a symptom s_i is believed to be shown by the patient ***patient***, under examination, with degree of belief r_1, i.e., *PatientDescription(patient,*s_i*)* = r_1, AND
  the system "knows" that whenever a patient shows the symptom s_i (**with degree of belief = 1**), then the patient is confirmed to show the symptom s_2, with degree of ***confirmation*** belief r_2, i.e., $R^c_{SS}(s_1, s_2)=r_2$, THEN
  it can be inferred that the patient ***patient***, under examination, shows the symptom s_2 with degree of belief r_3 such that:

$$r_3 = \text{MIN } [r_1, r_2].$$

- If more than one symptom confirm the same symptom with different degrees of belief, then the symptom with the greatest degree of confirmation belief is taken, to perform the inference composition. This inference rule is applied to infer for all symptoms for which it is meaningful.

The specification of this rule of inference is given by the following schema:

$COMPOSITION_4$

PatientSymptomPattern

KNOWNMANIFESTATION : $\mathbb{P}$ MANIFESTATION

KNOWNSYMPTOM : $\mathbb{P}$ SYMPTOM

R^4_{PS} : PATIENT $\times$ SYMPTOM $\rightarrowtail\!\!\!\!\!\!+$ FRACTION

KNOWNMANIFESTATION $\times$ KNOWNSYMPTOM $\subset$ dom R^c_{SS}

R^4_{PS} = PatientDescription $\circ$ ((KNOWNSYMPTOM $\times$ SYMPTOM) $\lhd$ R^c_{SS})

Rule 5. Symptom to symptom positive exclusion composition.

($COMPOSITION_5$)

- IF the patient ***patient*** shows a symptom s_1, with degree of belief r_1, i.e., *PatientDescription(patient,*s_1*)* = r_1, AND the system "knows" that whenever a patient shows the symptom s_1 **(with degree of belief = 1)**, then the patient shows also the symptom s_2, with a degree of ***confirmation*** belief r_2, i.e, $R^c_{SS}(s_1, s_2) = r_2$, THEN
  it can be inferred that the patient ***patient*** does ***not*** show the symptom s_2 with degree of belief r_3 such that:

$$r_3 = \text{MIN}\ [r_1, 1-r_2].$$

- If more than one symptom confirm the same symptom, then the symptom with the greatest degree of confirmation belief is taken to perform the inference composition. The inference rule is applied to infer for all symptoms for which it is

meaningful.

This rule can be specified by the following schema:

COMPOSITION$_5$

PatientSymptomPattern

KNOWNMANIFESTATION : $\mathbb{P}$ MANIFESTATION

KNOWNSYMPTOM : $\mathbb{P}$ SYMPTOM

R^5_{PS} : PATIENT $\times$ SYMPTOM $\leftrightarrow\!\!\!\!\!\rightarrow$ FRACTION

KNOWNMANIFESTATION $\times$ KNOWNSYMPTOM $\subset$ dom R^c_{SS}

R^5_{PS} = PatientDescription $\circ$ (KNOWNMANIFESTATION $\times$ KNOWNSYMPTOM) $\lhd$ (1-R^c_{SS})

Rule 6. Symptom to symptom negative exclusion composition.
(COMPOSITION$_6$)

- IF a symptom s_1 is believed to be shown by the patient, under examination, with degree of belief r_1, i.e.,
  PatientDescription(patient,s_1*)* = r_1, AND
  the system "knows" that whenever a patient shows the symptom s_1, then the patient shows the symptom s_2 with degree of *occurrence* belief r_2, i.e., $R^0_{SS}(s_1, s_2) = r_2$, THEN
  it can be inferred that the patient, under examination, does *not* show the symptom s_2 with degree of belief r_3 such that:

$$r_3 = \text{MIN}\ [1-r_1, r_2].$$

♦ If it is known that more than one symptom occur with the same symptom, then the symptom with greatest degree of occurrence belief is taken to perform this inference composition. This inference is applied to infer for all symptoms for which it is meaningful.

COMPOSITION$_6$

PatientSymptomPattern

KNWNMANIFESTATION : $\mathbb{P}$ MANIFESTATION

KNOWNSYMPTOM : $\mathbb{P}$ SYMPTOM

R^6_{PS} : PATIENT $\times$ SYMPTOM $\pfun$ FRACTION

KNOWNMANIFESTATION $\times$ KNOWNSYMPTOM $\subset$ dom R^o_{SS}

R^6_{PS} = (1-PatientDescription) $\circ$ ((KNOWNMANIFESTATION $\times$ KNOWNSYMPTOM) $\lhd$ R^o_{SS})

The three symptom-to-symptom inference compositions that we have specified above have the effect of inferring new degrees of belief for the presence of symptoms based on on degrees of belief of symptoms entered by the consulting physician. For some of the symptoms, COMPOSITION$_4$ may associate a degree of belief of 1. Those are confirmed diseases. We define ***confirmed symptoms*** in the following terms: a symptom s_2 is confirmed by a symptom s_1 if the symptom s_1 is shown by the patient, and if it is known that the presence of s_1 *always confirms* the presence of s_2. The specification for an operation that looks for confirmed symptoms, called *GetConfirmedSymptoms*, is given using the following schema:

GetCONFIRMEDSYMPTOMS

Ξ COMPOSITION$_4$

CONFIRMEDSYMPTOMS! : $\mathbb{P}$ SYMPTOM

$$CONFIRMEDSYMPTOMS! = \{ s : SYMPTOM \mid R^4_{PS}(patient, s) = 1 \}$$

For some of the symptoms, COMPOSITION$_5$ or COMPOSITION$_6$ will calculate degrees of belief of 1. Those are the excluded symptoms, i.e, symptoms for which it can be definitely concluded that they are absent in the patient. ***Excluded symptoms*** are recognized as stated in the following: a symptom s_1 excludes a symptom s_2 if s_1 is shown by a patient and if it is known that the presence of s_1 *never confirms* the presence of s_2 or if it is definitely established that the patient does not show the symptom s_1 and it is known that s_1 *always occurs* with s_2. The specification of an operation that looks for excluded symptoms is given by the following schema:

GetExcludedSymptom

Ξ COMPOSITION$_5$

Ξ COMPOSITION$_6$

EXCLUDEDSYMPTOMS! : $\mathbb{P}$ SYMPTOM

$$EXCLUDEDSYMPTOMS! = \{s : SYMPTOM \mid R^5_{PS}(patient, s) = 1\} \cup \{s : SYMPTOM \mid R^5_{PS}(patient, s) = 1\}$$

Confirmed as well as excluded symptoms are added to the

pattern of symptoms of the patient, using the following two operations, *AddExcludedSymptom* and *AddConfirmedSymptom*, defined on the schema *PatientSymptomPattern*:

AddConfirmedSymptom

ΔPatientSymptomPattern

symptom? : CONFIRMEDSYMTOMS

---

patient' = patient

MANIFESTATION = MANIFESTATION $\cup$ {symptom?}

PatientDescription' = PatientDescription $\oplus$ {(patient, symptom?) $\mapsto$ 1}

AddExcludedSymptom

ΔPatientSymptomPattern

symptom? : EXCLUDEDSYMPTOMS

---

patient' = patient

MANIFESTATION' = MANIFESTATION $\cup$ {symptom?}

PatientDescription' = PatientDescription $\oplus$ {(patient, symptom?) $\mapsto$ 0}

The operations *AddExcludedSymptom* and *AddConfirmedSymptom* make use of the override function operator (circle plus). The effect of this override function operator is to leave the function *PatientDescription* unchanged for symptoms that are not members of the class of inferred symptoms, to augment it to

include already unrecorded symptoms, and to modify it for inferred symptoms which are already recorded, by modifying their weights according to whether they are confirmed (weight=1) or excluded (weight=0).

4.3.3.3 Diseases-to-Disease Inferences

Like symptoms, in medicine some diseases ***always occur*** with some diseases and ***never occur*** with some others. Certain diseases ***always confirm*** some diseases and ***never confirm*** the presence of some others. In this system we take also this knowledge into account, and we specify the same inference compositions as for symptoms to symptoms inferences, or symptoms to diseases inferences by applied to infer from diseases to diseases.

Rule 7. Disease to disease confirmation composition.

(COMPOSITION$_7$)

♦ IF the patient, under examination, is believed to suffer from the disease d_1 with degree of belief r_1, AND
the system "knows" that whenever a patient suffers from the disease d_1 (**with degree of belief** = 1), then the patient is confirmed to suffer also from the disease d_2 with degree of ***confirmation*** belief r_2, i.e., $R^{\hat{}}_{DD}(d_1, d_2) = r_2$, THEN
it can be inferred that the patient is suffering from the disease d_2 with degree of belief r_3 such that:

$$r_3 = \text{MIN}\ [r_1, r_2].$$

- If more than one disease confirm the same disease with different degrees of belief, then the disease with the greatest degree of confirmation belief is taken to perform the inference composition. This inference rule is applied to infer for all diseases for which it is meaningful.

The specification for this rule of inference is given by the following schema:

$COMPOSITION_7$

$COMPOSITION_1$

$FOUNDDISEASE : \mathbb{P}\ DISEASE$

$KNOWNDISEASE : \mathbb{P}\ DISEASE$

$R^7_{PD} : PATIENT \times DISEASE \pfun FRACTION$

$\{patient\} \times FOUDDISEASE \subset dom\ R^1_{PD}$

$FOUNDDISEASE \times KNOWNDISEASE \subset dom\ R^c_{DD}$

$R^7_{PD} = R^1_{PD} \circ ((FOUNDDISEASE \times KNOWNDISEASE) \lhd R^c_{DD})$

Rule 8. Disease to disease positive exclusion composition.

($COMPOSITION_8$)

- IF the patient, under examination, is believed to suffer from the disease d_1, with degree of belief r_1, AND
  the system "knows" that whenever a patient suffers from the disease d_1 **(with degree of belief = 1)**, then it can be confirmed that the patient suffers also from the disease d_2, with degree of ***confirmation*** belief r_2, i.e, $R^c_{DD}(d_1, d_2)=r_2$, THEN

it can be inferred that the patient, under examination, does *not* suffer from the disease d_2 with degree of belief r_3 such that:

$$r_3 = \text{MIN}\ [r_1, 1-r_2].$$

- If more than one disease confirm the same disease, then the disease with the greatest degree of confirmation belief is taken to perform the inference composition. The inference rule is applied to infer for all diseases for which it is meaningful.

The specification of this rule of inference is the following:

COMPOSITION$_8$

COMPOSITION_1

$\text{FOUNDDISEASE} : \mathbb{P}\ \text{DISEASE}$

$\text{KNWNDISEASE} : \mathbb{P}\ \text{DISEASE}$

$R^8_{PD} : \text{PATIENT} \times \text{DISEASE} \leftrightarrow \text{FRACTION}$

$\{\text{patient}\} \times \text{FOUNDDISEASE} \subset \text{dom}\ R^1_{PD}$

$\text{FOUNDDISEASE} \times \text{KNOWNDISEASE} \subset \text{dom}\ R^c_{DD}$

$R^8_{PD} = R^1_{PD} \circ ((\text{FOUNDISEASE} \times \text{KNOWNDISEASE}) \lhd (1-R^c_{DD}))$

Rule 9. Disease to disease negative exclusion composition.

(COMPOSITION_3)

- IF the patient under examination is believed to suffer from the

disease d_1, with degree of belief r_1, i.e., AND
the system "knows" that whenever a patient suffers from the disease d_1 **(with degree of belief = 1)**, then the patient suffers also from the disease d_2, with degree of *occurrence* belief r_2, i.e., $R^{o}_{DD}(d_1, d_2) = r_2$, THEN
it can be inferred that the patient under examination does *not* suffer from the disease d_2, with degree of belief r_3 such that:

$$r_3 = \text{MIN}\ [1-r_1, r_2].$$

♦ If it is known that d_2 occurs with more than one disease, then the disease with the greatest degree of occurrence belief is taken to perform this inference composition. This inference is applied to infer for all diseases for which it is meaningful.

In the schema below, we specify this rule of inference;

COMPOSITION$_9$

COMPOSITION$_1$

FOUNDDISEASE : $\mathbb{P}$ DISEASE

KNOWNDISEASE : $\mathbb{P}$ DISEASE

R^{9}_{PD} : PATIENT $\times$ DISEASE $\pfun$ FRACTION

$\{patient\} \times FOUNDDISEASE \subset dom\ R^{1}_{PD}$

$FOUNDDISEASE \times KNOWNDISEASE \subset dom\ R^{o}_{DD}$

$R^{9}_{PD} = (1-R^{1}_{PD}) \circ ((FOUNDDISEASE \times KNOWNDISEASE) \lhd R^{o}_{DD})$

Now, using two operations, *GetExcludedDiseases* and

GetConfirmedDiseases, the system looks for confirmed and excluded diseases. The specifications of these two operations is given in the following two schemas:

GetConfirmedDisease

Ξ COMPOSITION$_7$

CONFIRMEDDISEASES! : $\mathbb{P}$ DISEASE

$$CONFIRMEDDISEASES! = \{d : DISEASE \mid R^{7}_{PD}(patient, d) = 1\}$$

GetExcludedDisease

Ξ COMPOSITION$_8$

Ξ COMPOSITION$_9$

Ξ COMPOSIYION$_2$

Ξ COMPOSITION$_3$

EXCLUDEDDISEASES! : $\mathbb{P}$ DISEASE

$$\begin{aligned} EXCLUDEDDISEASES! = {} & \{d : DISEASE \mid R^{8}_{PD}(patient, d) = 1\} \\ & \cup \{d : DISEASE \mid R^{9}_{PD}(patient, d) = 1\} \\ & \cup \{d : DISEASE \mid R^{2}_{PD}(patient, d) = 1\} \\ & \cup \{d : DISEASE \mid R^{3}_{PD}(patient, d) = 1\} \end{aligned}$$

Confirmed as well as excluded diseases are added to the pattern of diseases describing the patient under examination using the two operations specified by the following two schemas:

DIAGNOSTICS

$COMPOSITION_1$

$Differential\_diagnosis! : PATIENT \times DISEASE \rightarrow\!\!\!\!\!\!+ \; FRACTION$

$threshold : FRACTION$

$$Differential\_diagnosis! = \{(patient, disease) \mid R^1_{PD}(patient, disease) > threshold\} \lhd R^1_{PD}$$

CHAPTER 5

CONCLUSION

In the present thesis, the work was carried out in the purpose of investigating the ***specifiability*** of the decision making processes of an expert, that are usually emulated by computer programs in the form of expert systems. The aim is to develop a ***specification model*** of a medical expert system, using one of the standard formal methods for software development. The work is centered around one of the well-known model-based formal methods, developed in the last decade for the specification and subsequent development steps of large sequential software systems. This method is namely the ***Z-method.*** The work can be seen as consisting of three main parts:

In the first part, the problems that have led the software industry into a crisis situation, in the late 1960s, are retraced first. Then, the concept of software engineering, which was invented to solve the software crisis, is introduced. This is followed by a discussion of software qualities, which are the desirable qualities that any software system should possess, and which software systems developed before the advent of software engineering lacked. Next, the methods, both structured methods and formal methods, that drive software development in order to

integrate the principles of software engineering in the process of software systems development are reviewed. In this discussion of methods for software development, much emphasis is put on formal methods, stating their advantages as well as some of the problems which they currently find for their acceptance in the software industry. Tools as well as managements issues in software engineering are also briefly discussed in this first part.

The second part is a summary of the main knowledge representation techniques that have been developed and used for the design of practical expert systems. The discussion is limited to the declarative scheme of knowledge representation, because most of the best known systems, such as MYCIN, CASNET, and INTERNIST, have been developed using this formalism. In the declarative scheme of knowledge representation, the production rules technique, the semantic network technique, and the frames technique are discussed with a practical example of systems supporting each of them. Current research in the development of new techniques is also mentioned in this second part of the thesis.

In the third and last part of the thesis, the Z specification language is integrated in the formal system of fuzzy sets in order to describe a ***model*** of a ***rule-based*** expert system, intended to diagnose the acute abdominal pain emergency problem. The process of inference is based on three inference rules that have already been used for the development of a practical expert system, the CADIAG system. A method for the specification, using the Z specification language, of these inference rules is

proposed.

The development of the model that is proposed in this thesis has gone through two phases: the knowledge elicitation phase and the system modelling phase.

The knowledge elicitation phase consisted mainly of attendance to about ten meetings, of two hours each, with an expert surgeon in the diagnosis of the acute abdominal pain emergency. Through that series of meetings, we have been able to appreciate the essentials of the decision making processes involved in the diagnostic activity. Our experience in conducting this phase has permitted us to make the following important observation: *as expected*, it was not possible to communicate with the expert surgeon by using directly the notation of Z. This is due to the unfamiliarity of the expert surgeon with that notation. Furthermore, it appeared unrealistic to explain the notation for him. Indeed, we have used natural language (French) for the knowledge elicitation phase.

In the modelling phase, based on techniques found in the literature, the diagnostic knowledge collected during the first phase was encoded into the mathematical structures that the Z language offers to the specifier.

Although we have been able to develop a method for translating such vague concepts as those encountered in medicine into a mathematical model, we believe that the proposed specification lacks the description of some desirable features of expert systems. For example, we have not specified an explanation facility for the system, nor we have specified the questioning scheme of the user. Nevertheless, the method is there, and we are

convinced that the proposed specification can easily be augmented to include those aspects of the system.
The following remarks can be made, concerning further research in this topic:

1) We have pointed out earlier that we have conducted the knowledge elicitation activity using natural language. After encoding the knowledge into the Z notation, we used to retranslate that into natural language, and submit it to the expert surgeon for validation. However, in doing so, we believe that we missed a fundamental concept behind the use of formal methods in software development. In fact, one of the fundamental reasons for using a formal specification language in software development is that they allow the software developer to describe, with the user, ***precisely***, the system requirements, hence obtaining a basis for a contract that is signed by both participants (developer and user). Our specification lacks this property. We propose, thus, that the specification proposed be ***refined*** into a computer program, thus, continuing with the remaining steps of the system's life cycle. This will establish definitely the validity or discover the eventual defects of the proposed method.

2) As mentioned earlier in this thesis, the disease mechanisms involved in the acute abdominal pain emergency are not well understood. We believe that if the application area is carefully chosen then an interesting experiment would be to develop an expert system using a formal ***executable*** specification language.

The reason for suggesting an executable specification language for use is that it will allow the development of prototypes, thus permitting to explore confidently the knowledge domain.

BIBLIOGRAPHY

[1] SOMMERVILLE, I.: 'Software engineering', 1989, Third Edition, Addison-Wesley.

[2] SELL, P.S.: 'Expert systems: a practical introduction', 1985, Mc.Millan.

[3] BARR, A., FEINGEBAUM, E.A.: 'The handbook of Artificial Intelligence', Volume 1, 1982, Addison-Wesley, Inc.

[4] BARR, A., FEIGENBAUM, E.A.: 'The handbook of Artificial Intelligence', volume 2, 1982, Addison-Wesley.

[5] COHEN, P.R., FEIGENBAUM, E.A.: 'The handbook of Artificial Intelligence', Volume 3, 1982, Addison-Wesley.

[6] CORDIER, M.-O.: 'Les systemes experts', La Recherche Numero 151, 1984, **15**, pp. 60-70.

[7] CARROL, B.: 'Expert systems for clinical diagnosis: are they worth the effort?', Behaviourial Science, 1987, **32**, pp. 274-292.

[8] SHORTLIFFE, E.,H.: 'Computer-based consultations in clinical therapeutics: explanation and rule acquisition capabilities of the MYCIN system', Computers and Biomedical Research, 1975, 8, pp. 303-320.

[9] WEISS, S.M., KULIKOWSKI, C.A., AMAREL, S., and SAFIR, A.: 'A model-based method for computer-aided medical decision-making', Artificial Intelligence, 1978, 11, pp. 145-172.

[10] ADLASSNIG, K.-P., KOLARZ, G.: 'CADIAG-2: Computer-assisted medical diagnosis using fuzzy subsets', Approximate Reasoning in Decision Analysis, 1982, pp. 219-247.

[11] ADLASSNIG, K.-P., KOLARZ, G., W. SCHEITHAUER: 'Present state of the medical expert system CADIAG-2', Meth. Inform. Med., 24, 1985, pp. 13-20.

[12] DOJAT, M., BROCHART, L., LEMAIRE, F., and HARF, A.: 'A knowledge-based system for assisted ventilation of patients in intensive care units', International Journal of Clinical Monitoring and Computing, 1992, 9, pp. 239-250.

[13] FROHLICH, M.W., MILLER, P.L., and MORROW, J.S.: 'PATHMASTER: modelling differential diagnosis as "dynamic competition" between systematic analysis and disease-directed deduction', Computers and Biomedical Research, 1990, 23, pp. 499-513.

[14] MOORE, G.W., WAKAI, I., SATOMURA, Y., and GIERE, W.:

'TRANSOFT: Medical translation expert system', Artificial Intelligence in Medicine, 1989, 1, pp. 149-157.

[15] DE DOMBAL, F.T.: 'The diagnosis of acute abdominal pain with computer assistance: worldwide perspective', Ann. Chir, 1991, **45**, (4), pp. 273-277.

[16] POTTER, B., SINCLAIR, J., and TILL, D.: 'An introduction to formal specification and Z', 1991, Printice Hall International (UK), Ltd.

[17] JONES, C.B.: 'Systematic software development using VDM', 1986, Prentice-Hall International (UK) ltd.

[18] HOARE, C.A.R.: 'Communicating sequential processes', 1985, Prentice Hall International, UK, Ltd.

[19] HILLAL, D.K., SOLTAN, H.: 'To prototype or not to prototype? That is the question', Software Engineering Journal, November 1992, pp. 388-392.

[20] MACRO, A.: 'Software engineering: concepts and management', 1990, Printice Hall International (UK) Ltd.

[21] PRESSMAN, R.S.: 'Software engineering: a practitioner's approach', 1987, Second Edition, Mc Graw Hill.

[22] GHEZZI, C., JAZAYERI, M., MANDRIOLI, D.: 'Fundamentals of

software engineering', 1991, Printice-Hall International, Inc.

[23] SHORTLIFFE, E.H.: 'Consultation systems for physicians: the role of artificial intelligence techniques' in Readings in Artificial Intelligence, WEBBER, B.L., NILSSON, N.J. (Editors), 1981, Tioga Publishing Co., pp. 323-333.

[24] DAVIS, A.M.: 'Software requirements: analysis and specification', 1990, Prentice Hall International, Inc.

[25] PETERS, L.: 'Advanced structured analysis and design', 1988, Prentice Hall, Inc.

[26] DOWNS, E.: 'Structured Systems Analysis and Design Method: application and context', 1992, Second Edition, Prentice Hall International (UK) Ltd.

[27] YOURDON, E.: 'Modern Structured Analysis', 1989, Prentice Hall International.

[28] SUTCLIFFE, A.: 'Jackson System Development', 1988, Prentice Hall International (UK) Ltd.

[29] WOODCOCK, J.C.P., LOOMES, M.: 'Software engineering mathematics: formal methods demystified', 1988, Pitman.

[30] LUCKHAM, D.C., and VON HENKE, F.W.: 'An overview of Anna,

a specification language for Ada', IEEE Transactions on Software Engineering, 1985, 2, (2), pp. 9-22.

[31] BURSTALL, R.M., and GOGUEN, J.A.: 'An informal introduction to specifications using Clear' in BOYER, R.S., and MOORE, J.S. (Eds): 'The correctness problem in computer science', 1981, Academic Press, London, pp. 185-213.

[32] FUCHS, N.E.: 'Specifications are (preferably) executable', Software Engineering Journal, September 1992, pp. 323-334.

[33] HAYS, I.(Editor): 'Specification Case Studies', 1987, Prentice Hall International (UK) Ltd.

[34] BOWEN, J., STAVRIDOU, V.: 'Safety-critical systems, formal methods and standards', Software Engineering Journal, July 1993, pp. 189-209.

[35] PLAT, N., HATWIJK, J.V., and TOETENEL, H.: 'Application and benefits of formal methods in software development', Software Engineering Journal, September 1992, pp. 335-346.

[36] POLACK, F.: 'Integrating formal notations and systems analysis: using entity relationship diagrams', Software Engineering Journal, September 1992, pp. 363-371.

[37] GANE, C.: 'Computer-Aided Software Engineering: the methodologies, the products, and the future', 1990,

Prentice-Hall International, Inc.

[38] SCHWARTZ, J.T.: 'The practical and not-yet-practical in software engineering' in Computing Tools For Scientific Problem Solving, 1990, Academic Press Limited, pp. 23-38.

[39] WALZ, D.B., ELAM, J.J., and CURTIS, B.: 'Inside a software design team: knowledge acquisition, sharing, and integration', Communications of the ACM, 1993, **36**, (10), pp. 63-76.

[40] KUIPERS, B.: 'New reasoning methods for artificial intelligence in medicine', Int. J. Man-Machine Studies, 1987, **26**, pp. 707-718.

[41] MEZHOUD, Belkacem ,and AZNI, Mohamed: 'Formal specification of a diagnostic program for the acute abdominal pain', Modelling, Measurement and Control, C, Vol. 44, **2**, 1994, pp. 41-52.

[42] DE DOMBAL, F.T., LEAPER, D.J., STANILAND, J.R., Mc CANN, A.P., and HORROCKS, J.C.: 'Computer-aided diagnosis of acute abdominal pain', British Medical Journal, 1972, **2**, pp. 9-13.

[43] HORROCKS, J.C., McCANN, A.P., STANILAND, J.R., LEAPER, D.J., and DE DOMBAL, F.T.: 'Computer-aided diagnosis: description of an adaptable system, and operational experience with 2034 cases', British Medical Journal, 1972, **2**, pp. 5-9.

[44] TAYLOR, T.R.: 'The role of computer systems in medical decision-making' in Human Interaction With Computers, SMITH, H.T., and GREEN, T.R.G. (eds), 1980, Academic Press, Inc (London) ltd., pp. 231-266.

[45] SPIEGELHALTER, D.J., and KNILL-JONES, R.P.: 'Statistical and knowledge-based approaches to clinical decision-support systems, with an application in gastroenterology', Journal of Royal Statistical Society A, 1984, **147**, (1), pp.35-77.

[46] ZADEH, L.A.: 'Fuzzy sets', Information and control, 1965, 8, pp. 338-353.

[47] LAURITZEN, S.L., SPIEGELHALTER, D.J.: 'Local computations with probabilities on graphical structures and their application to expert systems', J. R. Statist. Soc., B, 1988, **50**, (2), pp. 157-224.

[48] JENSEN, F.V., OLESEN, K.G., ANDERSEN, S.K.: 'An algebra of bayesian belief universes for knowledge-based systems', Networks, 1990, **20**, pp. 637-659.

[49] JENSEN, F.V., LAURITZEN, S.L., OLESEN, K.G.: 'Bayesian updating in causal probabilistic networks', Computational Statistics Quarterly, 1990, **4**, pp. 269-282.

[50] KIM, J.H., PEARL, J.: 'A computational model for causal and diagnostic reasoning in inference systems' Proceedings of

the 8th International Joint Conference on Artificial Intelligence, 1983, pp. 190-193.

[51] ANDREASSEN, S., JENSEN, F.V., OLESEN, G.: 'Medical expert systems based on causal probabilistic networks', Int. J. Biomed. Comput., 1991, pp. 1-30.

[52] TODD FRCS, B.S.: 'A model-based diagnostic program', Software Engineering Journal, May 1987, pp. 54-63.

[53] SPIVEY, J.M.: 'The Z notation: a reference manual', 1992, Prentice Hall International (UK) Ltd.

[54] SPIVEY, J.M.: 'An introduction to Z and formal specification', Software Engineering Journal, 1989, pp. 40-50.

[55] WOODCOCK, J.C.P.: 'Structuring specifications in Z', Software Engineering Journal, 1989, pp. 51-66.

[56] SANCHEZ, E.: 'Medical diagnosis and composite fuzzy relations', North-Holland Publishing Co., 1979, pp. 437-444.

APPENDIX A

THE Z NOTATION

A.1 INTRODUCTION

The Z notation [53] (also see [54] for a simple introduction) has been developed at the beginning of the last decade at the university of Oxford. Z is a model-based method, essentially based on predicate calculus and strongly typed set theory, which was primarily designed for the specification of large software systems. It allows software developers to describe formally the behaviour of software systems and to reason mathematically about them. Since Z is based on mathematics, its semantics is precisely defined. Thus, users of Z have the opportunity to write unambiguous and abstract specifications to describe *what* will be the behaviour of a software system. Some large projects have already made use of Z and it was claimed that appreciable benefits have been gained (see, for example, the IBM transaction processing system [37]). We have designed the present appendix to illustrate the Z notation, and to see how we can make use of it to describe a simple system, the password system of the VAX minicomputer. However, we do not claim that the following specification is a complete description of the VAX password

system.

A.2 STRUCTURE OF THE Z LANGUAGE

The Z language is a language of specification. As such it has a well defined *vocabulary* and *semantics*. The vocabulary of Z consists essentially of mathematical structures, such as sets, relations, functions, and sequences, which are well known in the mathematical theory of sets. The semantics (meaning) of each structure is formally defined. The set of all structures and the operations on them, that are used in Z constitutes the mathematical tool-kit of the language. The complete mathematical tool-kit as defined in the standard notation of Z, can be found in [53]. We will not discuss here these structures, but we will focus on some conventions which are proper to the language.

A.3 THE PASSWORD SYSTEM

We start by introducing the idea of *primitive sets*. Primitive sets are the basic sets whose objects nature is unimportant from the specification point of view (from the implementation point of view, this might not be the case). For the specification of the password system, we will need to speak about users' *names* and *passwords*. The system will sometimes issue messages. Let us say that users' names are drawn from a certain set NAME, and that users use passwords drawn from a set PASSWORD to access the system. The kind of objects contained in the sets NAME, PASSWORD is not important for our specification. In Z, we

write

[NAME, PASSWORD];

to say that NAME and PASSWORD are primitive sets.

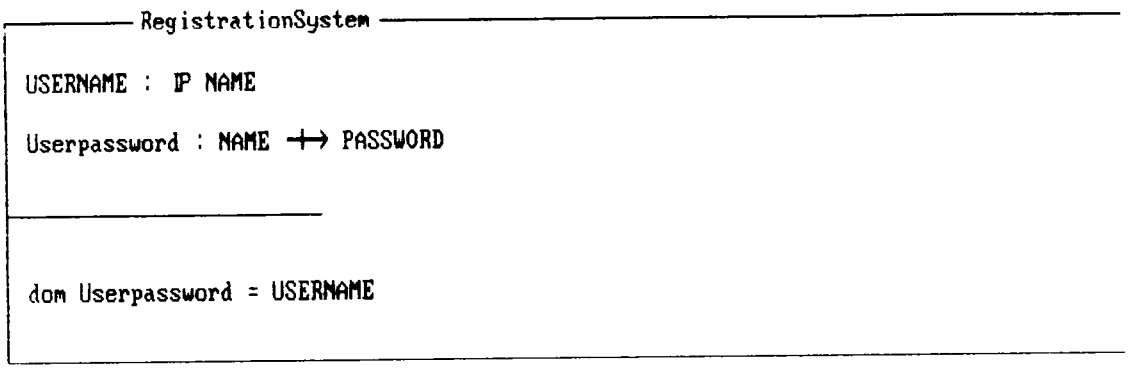
When the objects in a primitive set can be enumerated, we can use the so called *free type* definition to define that set. For example, in the description that follows, we shall see that the password system can issue to the user either of the two messages: "$" or "User Authorization Failure". Thus, if we call MESSAGE the set of these two messages, then we can use the free type definition to introduce the set MESSAGE.

MESSAGE ::= $ | UserAuthorizationFailure.

The most important structure of the Z language is the *schema*. A schema is a mathematical structure which introduces one or more variables and the constraints between their values. We illustrate the idea of a schema by using it to specify a portion of the password system, its registration system.

The registration system is a subsystem of the password system which permits the manager of the computer center to keep a list of users who are allowed to use the VAX system. Through it, the manager can register a new user, or he can suspend an already registered user. Each user is identified by his *username* and *password*. A user can have *only one* password. A given password can be owned by *more than one* user. A user can access the system by typing in his username and password. The registration system

is then able to recognize whether the user is allowed to use the system or not. It is sufficient to use a *set* and a *function* to capture the information needed to describe a model of this system. The state space description of the registration system can be written compactly using a schema:



As shown above, a schema consists of a signature part and a predicates part, separated by the cental dividing line. A schema may or may not have a name. The signature part is the part which is above the dividing line, and the predicates part is below the line. The signature part introduces *typed* variables. The predicates part specifies the *constraints* between the values of the declared variables. There can be more than one predicate line. Between each predicate line in the predicates part, there is an implicit logical AND. If desired this logical AND can be explicitly written.

The signature part of the schema describing the registration system contains declarations for two variables: *USERNAME* which is of type *set* and *Userpassword* which is of type *function*. The convention to show the type of a variable is to write the

variable followed by a *semicolon* then the *type of the variable*.

REMARK

Basic types are the primitive sets. *Composite types* can be obtained by combining basic types, using the well known operators of set theory. For example, we can see explicitly above that *USERNAME* is an element of the powerset (set of subsets) of the set *NAME*. This is sufficient to see that *USERNAME* is itself a set.

The predicates part in the state space description of the registration system says that the domain of the function *Userpassword* is the same as the subset *USERNAME*. This property is an *invariant* of the system, which means that the predicate of equality must be true under all states of the registration system.

For more illustration about the notation used in the Z language, we consider two *operations* on the registration system. Again, we use a schema to describe the operation of adding a new user to the list of users who are allowed to use the VAX system:

Add_User

Δ RegistrationSystem

username? : NAME

password? : PASSWORD

Userpassword ' = Userpassword $\cup$ { username? $\longmapsto$ password? }.

Some new notation appears in the above schema. The DELTA symbol indicates to the reader of the specification that the operation *Add_User* changes the state of the system. The operation needs two inputs: *username?* and *password?*. The convention is to decorate inputs with a "?". To show the effect of the operation *Add_User* on the system, the predicates part of the schema above shows the values of the variable *Userpassword* after the operation. In Z, the values of the variables after an operation are *decorated* with "'". Variables' values before the operation are left undecorated. In the schema above, the function *Userpassword* is augmented to include the name and password of the new user after the operation *Add_User*.

The schema describing the operation *add_User* can be equivalently written as in the following schema:

Add_User

$$USERNAME : \mathbb{P}\ NAME$$

$$USERNAME' : \mathbb{P}\ NAME$$

$$Userpassword : NAME \pfun PASSWORD$$

$$Userpassword' : NAME \pfun PASSWORD$$

$$username? : NAME$$

$$password? : PASSWORD$$

$$dom\ Userpassword = USERNAME$$

$$dom\ Userpassword' = USERNAME'$$

$$Userpassword' = Userpassword \cup \{username? \mapsto password?\}.$$

As can be seen in the above schema, the DELTA symbol is used

to tell the reader of the specification that: the variables declared for the system before and after an operation are merged, in the signature part of the operation schema, with the variables that the operation itself introduces, and that the constraints in the predicate part are also merged.

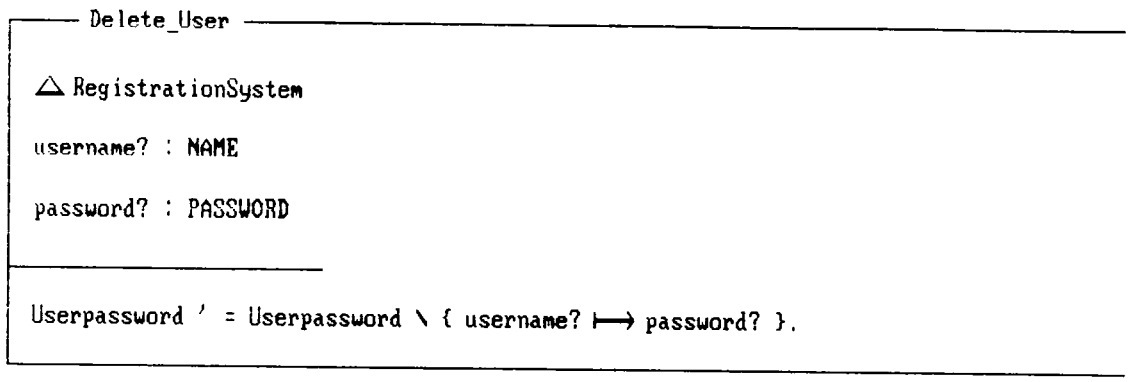
We can now use the specification of the operation *Add_user* to prove that USERNAME'= USERNAME U {username?}. The proof is as follows:

(1) Userpassword'= Userpassword U {username? -⇸ password?}. (specification of Add_User)
(2) dom Userpassword'= dom (Userpassword U {username? -⇸ password?}). (property of dom).
(3) dom Userpassword'= dom Userpassword U dom {username? -⇸ password?}. (property of dom).
(4) dom Userpassword'= USERNAME'. (specification of RegistrationSystem)
(5) dom Userpassword = USERNAME. (specification of RegistrationSystem)
(6) dom {username? -⇸ password?} = {username?}. (property of dom)
(7) USENAME'= USERNAME U {username?}. (substitution of (4), (5), and (6) in (3)).

formal proofs like the one given above are very useful when specifying software systems. They are a means for keeping conciseness of specifications and they allow to check for their consistency.

A second operation on the registration system can be that

of suspending an already registered user. A schema is used to describe it:



Like the operation Add_user, the operation Delete_User changes the state of the system. Its effect is to remove the name of the user to suspend from the list of allowed users. The symbol "\" is used to say that the object *{username? ↛ password?}* is subtracted from the function *Userpassword*.

Up to now, we have described only a portion of the password system. Now we describe another portion of the system; a subsystem that keeps track of the lists of currently logged in and logged out users. We call this subsystem *WorkingSystem*.

A user of the system cannot be logged in and logged out at the same time. The union of the set of logged in users and the set of logged out users forms the complete set of registered users. The state space description of the *WorkingSystem* is given in the schema below:

$$
\begin{array}{|l}
\hline
LOGGEDINUSER : \mathbb{P}\ NAME \\
LOGGEDOUTUSER : \mathbb{P}\ NAME \\
USERNAME : \mathbb{P}\ NAME \\
\hline
LOGGEDINUSER \cap LOGGEDOUTUSER = \varnothing \\
LOGGEDINUSER \cup LOGGEDOUTUSER = USERNAME \\
\hline
\end{array}
$$

Now we merge the two subsystems described above to describe the state space of the password system. We call it *PasswordSystem:*

$$
\begin{array}{|l}
\hline
\textbf{PasswordSystem} \\
\hline
RegistrationSystem \\
WorkingSystem \\
\hline
\end{array}
$$

The schema describing the password system can be written using a single detailed schema as follows:

$$
\begin{array}{|l}
\hline
\textbf{PasswordSystem} \\
\hline
LOGGEDINUSER : \mathbb{P}\ NAME \\
LOGGEDOUTUSER : \mathbb{P}\ NAME \\
USERNAME : \mathbb{P}\ NAME \\
Userpassword : NAME \pfun PASSWORD \\
\hline
LOGGEDINUSER \cup LOGGEDOUTUSER = \varnothing \\
LOGGEDINUSER \cup LOGGEDOUTUSER = USERNAME \\
dom\ Userpassword = USERNAME \\
\hline
\end{array}
$$

Thus, we have described each portion of the password system separately and then we have combined them to describe the whole system. This technique of presenting a specification piece by piece then combining the pieces is particularly helpful when we deal with large systems specifications [55].

Now, we consider one operation on the password system to illustrate more notation. The operation is called *login*. It permits a user to access the system. To access the system, a user types in his username and password. If the information which is entered is correct then he can access the system, otherwise he will be refused the access. Here is what happens in the case where the user enters correct information:

Login

Δ PasswordSystem

username? : NAME

password? : PASSWORD

---

username? $\in$ USERNAME

Userpassword(username?) = password?

LOGGEDINUSER ' = LOGGEDINUSER $\cup$ { username? }.

Below, we specify an operation that produces a message in the case where the user attempts to enter the system by supplying incorrect information:

UserAuthorizationFailure

Ξ PasswordSystem

username? : NAME

password? : PASSWORD

message! : MESSAGE

---

username? $\notin$ USERNAME $\vee$ Userpassword(username?) $\neq$ password?

message! = " User Authorization Failure ".

The new notations that appears in the above schema are: the "XI" symbol and the "!". The "XI" symbol is used to describe an operation which does not change the state of a system. The "!" symbol is used to decorate output values that an operation returns. Thus, if the user enters incorrect information for the system, the system state will not change, and it responds by issuing a message telling the user that he is not authorized.

We can now describe a robust version (we call it *RLogin*) of the login operation which take into account both the case where the user supplies correct information and the case where he supplies erroneous information. But first, we describe an operation which produces the message "*$*". Here is the description for this simple operation:

Success

message! : MESSAGE

---

message! = " $ "

The description for the operation *RLogin* is:

$$RLogin \triangleq (Login \wedge Success) \vee UserAuthorizationFailure$$

The description above, of the operation RLogin, is written using two operators from the *schema calculus*: the logical AND and the logical OR. The schema calculus is very useful for describing large system specifications. It permits a structuring mechanism by which the specifier can describe separately many aspects of the system, then using those structures to describe at the end the entire system.

To see what ANDing or ORing schemas means we rewrite the specification for RLogin using a detailed schema as follows:

RLogin

Δ PasswordSystem

username? : NAME

password? : PASSWORD

message! : MESSAGE

(username? $\in$ USERNAME $\wedge$ Userpassword(username?) = password? $\wedge$

message! = "$") $\vee$

(username? $\notin$ USERNAME $\vee$ Userpassword(username?) $\neq$ password? $\wedge$

message! = " User Authorization Failure ").

APPENDIX B

FUZZY SETS

B.1 INTRODUCTION

An ordinary set, as defined in conventional set theory, is a useful model for classifying objects having some well defined properties. However, we often encounter situations where we need to speak about properties based on subjective judgements that we assign to objects in the real world. In those situations, ordinary sets are, unfortunately, not suitable as models for the classes of such objects. The purpose of the theory of fuzzy sets developed by ZADEH [46] was to provide a framework within which imprecise concepts can be captured and treated mathematically. In this appendix, we summarize the important properties of fuzzy sets.

B.2 DEFINITIONS

Given an ordinary reference set E, a fuzzy subset A of the reference set E is characterized by a membership function $\mu_A(e)$ which associates with each object e in E a real number in the interval [0, 1]. The value $\mu_A(e)$ indicates the grade of membership of e to A. The nearer $\mu_A(e)$ is to unity the higher the grade of membership of e to E. Clearly the characteristic

function of an ordinary set can take only two values: 1 or 0, translating the logic values TRUE or FALSE, and this according to whether e belongs to A or not.

Example

In medicine, the concept of illness is not precise. The boundaries of the class of ill persons with respect to the class of healthy persons are not well defined. Thus, the class of ill persons is a fuzzy subset in the reference set of all individuals. If p_1 and p_2 are two individuals we can write

$\mu_{ill}(p_1)=0.9$

$\mu_{ill}(p_2)=0.2$

B.3 OPERATIONS ON FUZZY SETS

The usual operations of inclusion, union, intersection, and complementation which are defined for ordinary sets are also defined for fuzzy sets. These are defined below.

Let E be a reference set. Let A and B be two fuzzy subseys in the reference set E and $\mu_A(e)$ and $\mu_B(e)$ their respective characteristic functions. Then we define

Inclusion. The fuzzy subset A is included in the fuzzy subset B if and only if

$\forall e \in E,\ \mu_A(e) < \mu_B(e)$

Intersection. The intersection of the fuzzy subsets A and B is also a fuzzy subset in E denoted by A∩B whose characteristic function is given by

$$\forall e \in E,\ \mu_{A \cap B}(e) = MIN\ [\mu_A(e), \mu_B(e)]$$

Union. The union of the fuzzy subsets A and B is also a fuzzy subset in E denoted by AUB whose characteristic function is given by

$$\forall e \in E,\ \mu_{A \cup B}(e) = MAX\ [\mu_A(e), \mu_B(e)]$$

Complementation. The complement of the fuzzy subset A is fuzzy subset in E denoted by A' whose characteristic function is given by

$$\forall e \in E,\ \mu_{A'}(e) = 1 - \mu_A(e)$$

B.4 FUZZY RELATIONS

Let E and E' be two ordinary reference sets. A fuzzy relation R from E to E' is a fuzzy subset in the cartesian product E x E'. R is characterized by a membership function $\mu_R(e,e')$ which associates to each ordered pair (e,e'), respectively in E and E', a real number in the interval [0,1]. The characteristic function of R captures the concept that e is more or less related to e'.

$$\mu_R(e,e') : E \times E' \rightarrow [0,1].$$

Composition of fuzzy relations

Let E, F, and G be three ordinary reference sets, and let R, and S be two fuzzy relations defined respectively on E x F and F x G.

$$\mu_R : E \times F \rightarrow [0,1]$$

$$\mu_S : F \times G \rightarrow [0,1].$$

Then the composition of the fuzzy relation R with the fuzzy relation S is a fuzzy relation in E x G whose characteristic function is given by

$$\forall e \in E,\ \forall g \in G,\ \mu_{SoR}(e,g) = MAX_f\ MIN\ [\mu_R(e,f), \mu_S(f,g)].$$

We have now reviewed the important concepts of fuzzy sets. Other properties such as covexity of a fuzzy set exist, and can be found in [46]. However, since we did not make use of these further properties in this thesis, we do not discuss them in this appendix. Fuzzy sets have been introduced in medicine for the first time by SANCHEZ [56]. In particular, the concept of fuzzy relation has proved to be very useful for formalizing the approximate resoning that is frequently encountered in this area.

A N N E X E

Liste et composition du jury en vue de la soutenance de mémoire de Magister en Ingineerie des Systèmes Electroniques par Mr AZNI Mohamed.

PRESIDENT /	Mr . BOULARAS	(Pr - U.S.T.H.B).
RAPPORTEURS/	Mr HOLCOMBE	(Pr - U. Sheffield (U.k).
	Mr B. MEZHOUD	(MA. Master - INELEC)
EXAMINATEURS /	Mr M. DjEDDI	(PhD.CC - INELEC).
	Mr H. AZZOUNE	(T.U/CC - U.S.T.H.B).