EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Eindhoven University of Technology

MASTER

Implementing a critiquing system to provide decision support in the ICU : the CritICIS system

de Clercq, P.A.

Award date:
1996

Link to publication

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

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

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

Eindhoven University of Technology
Department of Electrical Engineering
Division of Medical Electrical Engineering

Implementing a critiquing system to provide decision support in the ICU

the CritICIS system

By P.A. de Clercq

M.Sc. thesis
Carried out from January 1996 to October 1996
In order of:
Dr. Ir. P. Cluitmans
Under supervision of:
Dr. Ir. J.A. Blom

The department of Electrical Engineering of the Eindhoven University of Technology accepts no responsibility for the contents of M.Sc. theses or reports on practical training periods.

Abstract

Due to the development of new technology for diagnostic and therapeutic purposes, combined with the introduction of microprocessor technology, the amount of data collection in Intensive Care Units (ICUs) has increased enormously. In order to collect, store and manage this flow of information, Patient Data Management Systems (PDM-Systems) were introduced. An example of such a system is the Intensive Care Information System (ICIS). ICIS is a PDMS, designed to process data at the ICU of the Catharina hospital, situated in Eindhoven, the Netherlands.

To improve the quality of the data, stored in ICIS, a critiquing system was implemented to provide decision support to the users of ICIS. This system, called CritICIS, accepts a medical protocol as well as data from the ICIS database and verifies 1) the consistency of the database itself and 2) the consistency of certain treatments (e.g., it is recommended not to administer penicillin to a patient that is allergic to penicillin). CritICIS is implemented using the SIMPLEXYS toolbox, a set of tools to design real time expert systems and Borland Delphi, a Rapid Application Development (RAD) tool. Although implemented as a separate application, CritICIS acts as an integrated part of ICIS: the user selects a patient in ICIS, after which ICIS activates the critiquing system. The selected patient's data is then gathered and processed by CritICIS and possible inconsistencies are shown as warnings to the user. The system is currently in the development phase and is now tested by the medical staff. The first results are promising: as a result of the integrated user interface of CritICIS as well as the utilized critiquing approach, the users of ICIS are satisfied with the critiquing system. Therefore, the conclusion is drawn that it is possible to successively implement a critiquing system that provides decision support to the medical staff of the above-mentioned ICU.

This paper describes the development of the CritICIS system. It provides information about the problem domain, ICIS and the SIMPLEXYS toolbox as well as the internal functioning of CritICIS, it's means of communication with (the users of) ICIS and the knowledge acquisition process.

Acknowledgments

First, I would like to thank the people at the ICU of the Catharina hospital for their hospitality and for allowing me to use their facilities, especially Jan v.d Berk and Hannie Megens for testing CritICIS and providing me with invaluable feedback.

Furthermore, I would like to thank Onno van Zinderen from INAD, for providing information about ICIS and for helping me with some problems, regarding Delphi and Access.

Several people from the section Medical Electrical Engineering assisted me during the development of CritICIS. I would like to thank Hennie v.d. Zanden for general support, Sjoerd Ypma for helping me with some network problems and Harrie Kuipers for answering several SQL questions and for suggesting an appropriate format for the CritICIS database.

I would especially like to thank Luc Cluitmans for answering all my Delphi related questions and more. I enjoyed working together with him.

Last, but not least, I would like to thank Erik Korsten from the ICU for supplying various rules and for his enthusiastic comments and suggestions to improve CritICIS, and especially Hans Blom, for his guidance and support during my theses work.

Contents

1. Introduction — *1*

1.1. Patient monitoring — **1**

1.2. Decision support based on computer-stored medical records — **2**

1.3. Overview of the text — **4**

2. Expert systems in medicine — *6*

2.1. From computer understanding to knowledge representation — **6**
- 2.1.1. The early years — 6
- 2.1.2. Expert systems — 7

2.2. Components of an expert system — **8**

2.3. Production systems — **10**
- 2.3.1. Syntax — 11
- 2.3.2. Communication between rules — 11

2.4. Real time expert systems — **13**
- 2.4.1. Expert systems operating in dynamic environments — 13
- 2.4.2. The difference between traditional and real time expert systems — 14

2.5. Expert systems in medicine — **16**
- 2.5.1 Critiquing systems — 16
- 2.5.2. Exploring the critiquing approach — 17

3. The Intensive Care Information System (ICIS) — *19*

3.1. Patient Data Management Systems (PDM-Systems) in intensive care — **19**
- 3.1.1. The pros and cons of a PDMS — 20
- 3.1.2. Classifications of PDM-Systems — 22

3.2. ICIS — **22**

3.3. The monitoring equipment — **27**
- 3.3.1. The SDN Monitoring network — 28
- 3.3.2 The ICIS graphics server — 29
- 3.3.3. Current stage in development — 29

4. Development tools and techniques — *30*

4.1. Developing clinical systems using rapid prototyping — **30**
- 4.1.1. Evolutionary rapid prototyping — 31
- 4.1.2. The path from prototype to application — 32
- 4.1.3. Developing a critiquing system by means of rapid prototyping — 33

4.2. The Borland Delphi programming environment — **34**
- 4.2.1. Component technology — 34
- 4.2.2. Visual development environments — 36
- 4.2.3. Building applications by means of Delphi — 36

4.3. The SIMPLEXYS toolbox — **37**
- 4.3.1. The SIMPLEXYS programming language — 37
- 4.3.2. The SIMPLEXYS rule compiler and extensions — 41
- 4.3.3. The SIMPLEXYS inference engine — 44
- 4.3.4. Adapting the SIMPLEXYS inference engine to object-oriented environments — 45

5. Critiquing ICIS: The development of CritICIS *47*

5.1. An introductory example **47**

5.2. An overview of CritICIS **50**
5.2.1. The user interface 51
5.2.2. The data acquisition components 52
5.2.3. The SIMPLEXYS module 56

5.3. Executing the critiquing system **61**
5.3.1. Initializing CritICIS 61
5.3.2. Activating CritICIS 62
5.3.3. Processing the warnings 69
5.3.4. Ending CritICIS 72

5.4. Structure of the knowledge base **72**
5.4.1. Warning classification 73
5.4.2. Utilizing the context number 75
5.4.3. Executing warnings through strategic rules 76
5.4.4. Adding warning records by means of THELSEs 76

5.5. Developing CritICIS by means of rapid prototyping **77**
5.5.1. Developing the CritICIS module 77
5.5.2. Developing the knowledge base 78

6. Conclusions *80*

REFERENCES *83*

APPENDIX A: The knowledge base of CritICIS *85*

APPENDIX B: Developed Delphi components *91*

1. Introduction

1.1. Patient monitoring

Patient monitoring is an important process in specialized hospital departments such as the *Intensive Care Unit* (ICU), where the patient's condition is often very dynamic and unstable. Patient monitoring is a complex process that includes observation (measurement), interpretation, evaluation (diagnoses) and control (therapy) of the patient's condition (figure 1.1).

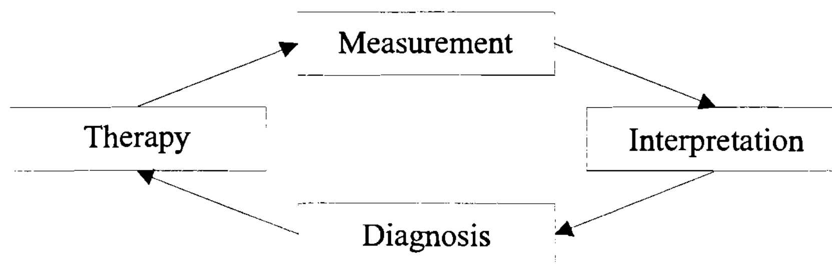

Figure 1.1: The patient monitoring process, adapted from [Reader Met. i.d Gen. II]

The first step in the patient monitoring process is the measurement of the patient's physiological variables. Examples of these variables are:

- the electrocardiogram (ECG)
- the patient's blood pressure
- the patient's temperature

The treating physicians interpret and evaluate these variables and suggest a suitable therapy. The patient's response on this therapy is monitored again, so that it is possible to make alterations or adjustments if necessary.

Patient monitoring is a continuous process. It is essential that the acquisition of the physiological variables takes place fast and accurate in order to keep the patient's condition stable. This aspect of patient monitoring is called *real time* monitoring. Real time monitoring is crucial in hospital departments such as the ICU (Intensive Care Unit).

1.2. Decision support based on computer-stored medical records

When monitoring a patient, a number of physiological variables are measured. The number of measured variables depends on the patient's condition; a patient that undergoes a surgical operation requires more monitoring than a recovering patient. Other aspects, such as the patient's age or medical history and the fact that some measurement methods are more expensive than others also determine the quantity of measured variables.

In the past, these variables were written down on paper. Every patient had a form, containing the patient's data. Apart from the physiological variables, the form also included personal information, such as the height, weight and religion of the patient as well as comments from the treating physicians and the medical staff.

In addition to these paper forms, patient data is nowadays also stored in computer records, forming a *medical database*. An example, taken from the Intensive Care Information System (ICIS), is shown in table 1.1.

Patient-number	Admission-date	Admission-time	Room-number	Discharge-date	Urgent admission
51165635	1996-01-28	01:56:39	2	1996-01-28	FALSE
123456789	1996-06-11	15:58:33	3		FALSE
2053793518	1996-01-24	11:15:09	1	1996-01-25	FALSE
2083279510	1996-01-24	18:59:49	3	1996-01-25	FALSE
3024670526	1996-01-25	16:07:13	1	1996-01-26	FALSE
4056983510	1996-01-28	19:03:32	3	1996-01-28	TRUE
4106993522	1996-01-25	13:26:06	1	1996-01-26	FALSE
4113543529	1996-01-29	13:42:41	2		TRUE
5092910519	1996-01-23	14:35:50	4	1996-01-25	FALSE
6032333018	1996-01-25	15:27:26	2	1996-01-28	FALSE

Table 1.1: A (small) section of the ICIS database

In many cases, the monitoring equipment is connected to the medical database. Because of this, the data obtained from the monitoring equipment is stored directly into the database without human intervention (figure 1.2)

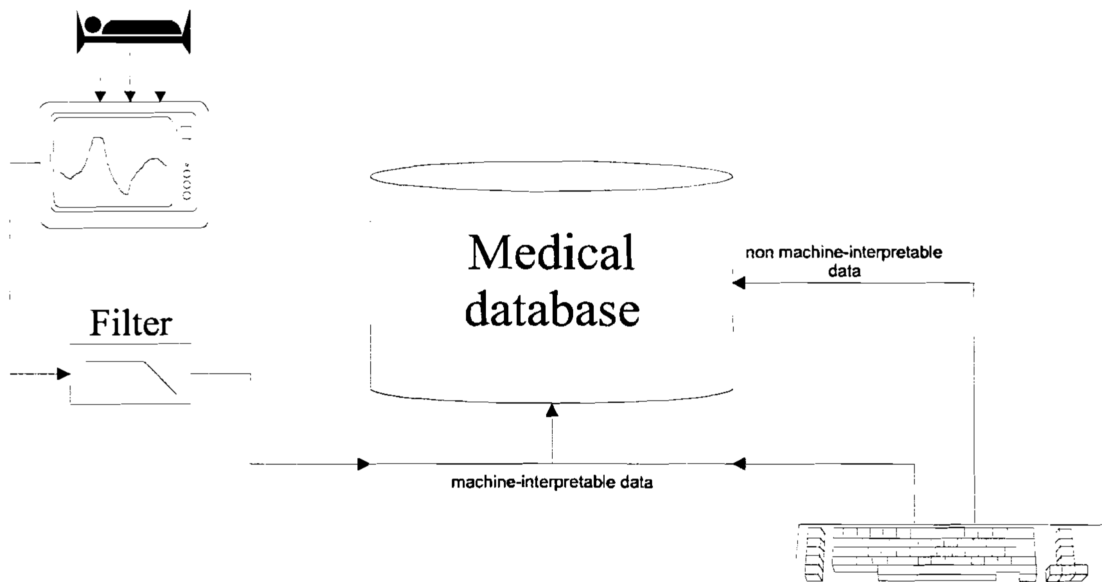

Figure 1.2: Information sources of the medical database

The amount of transferred data from the monitor to the database varies with the measured physiological variable (for example, the patient's blood pressure is sometimes measured once every 5 minutes, whereas the patient's ECG is measured continuously). Especially regarding variables that are measured continuously, a filter is usually placed between the monitor equipment and the database to reduce the amount of (real time) data. The output of the filter usually consists of average values, calculated between two points of time.

The medical database contains not only the monitoring data, but also data from the treating physicians and the medical staff. This data classifies into 2 categories:

- machine-interpretable data: the data in this category is interpretable by an automaton (computer) and it is possible to use it in calculations. Often the data in this category consists of numbers (such as the patient's age or height), but sometimes also text (for example, the name of a prescribed medicine). Usually in the second case, the text is internally encoded into a numerical value or code. Note that the data from the monitoring equipment also falls into this category.
- non machine-interpretable data: data in this category usually consists of so-called 'free text'; comments about the patient and/or the patient's environment, such as

phone numbers of relatives. A computer cannot process this data or use it in calculations.

Using this (machine-interpretable) data, one could ask the question if it is possible to construct an automaton that supplies some sort of decision support to the treating physicians. Research showed that this question can be answered in the affirmative [Van Der Lei, 1991]. Based on a model of the patient, a so-called *critiquing system* is able to supply critique, using data from a medical database.

Critiquing systems are usually realized as *expert systems*. The research of expert systems derives from the longer existing research of *Artificial Intelligence* (AI). The main difference between expert systems and 'traditional' AI-programs, is that expert systems tend to concentrate on narrow domain problem areas, whereas AI research tries to solve a larger class of problems (using so-called *general problem solvers*). Because critiquing systems are narrow domain area experts, they fall into the same class as expert systems.

1.3. Overview of the text

This paper describes the development of a critiquing system, called CritICIS. CritICIS is able to critique the actions of the treating physicians and the medical staff at an ICU, situated at the Catharina hospital in Eindhoven, the Netherlands. In the ICU, patient data is collected and stored into a medical database, using a Patient Data Management System (PDMS), called ICIS (Intensive Care Information System). CritICIS has access to the medical database and critiques the user, based on the obtained data from ICIS.

Chapter 2 presents a general overview of (real time) expert systems and critiquing systems in medicine. **Chapter 3** gives some background information about the problem domain. It describes ICIS and it's environment, such as the monitoring equipment. **Chapter 4** describes the tools and techniques, used in the development process. The internal structure of CritICIS and its functioning is described in **chapter 5**, along with the structure of the knowledge base of CritICIS. The knowledge base

contains the expert knowledge that CritICIS needs in order to critique the actions of the treating physicians. Finally, this chapter provides information about the knowledge acquisition process (the elicitation and encoding of the expert's knowledge into the knowledge base). The last chapter, **chapter 6**, contains results, conclusions and further recommendations.

2. Expert systems in medicine

2.1. From computer understanding to knowledge representation

The research of expert systems is relatively new. It derives from the longer existing research of Artificial Intelligence (AI), which is often described by [Barr and Feigenbaum, 1981]:

> *AI is the part of computer science concerned with designing intelligent computer systems, that is, systems that exhibit the characteristics we associate with intelligence in human behavior - understanding language, learning, reasoning solving problems, and so on.*

2.1.1. The early years

The history of AI (Artificial Intelligence) started in the 1950s, when AI researchers wrote symbolic programs to solve problems that normally required human intelligence. These programs were called general problem solvers and they were able to solve puzzles and prove simple theorems. In the 1960s and 1970s, the emphasis in AI research shifted from general problem solving to the development of computational models of human intelligence, including both its cognitive and perceptual aspects [Duda and Shortliffe, 1983]. The belief that a few reasoning laws (similar to human reasoning) could produce intelligent behavior, dominated this period. People were asked to think aloud when they solved a problem, after which the AI researchers tried to analyze and formalize the used reasoning strategies. *Computer understanding* and *heuristic search* were the main research topics in this period.

But in the last two decades it turned out that this approach would not lead to any breakthroughs, mainly because the problem independent reasoning methods were generally too weak to solve complex individual problems. It became clear that expertise implies more than utilizing the public knowledge and strategies, known to AI programs [Hayes-Roth *et al.*, 1983]. Human experts possess various kinds of

knowledge, such as clarifying the problem, judging the reliability of facts and deciding whether a solution is reasonable [Duda and Shortliffe, 1983].

2.1.2. Expert systems

Disappointed with general problem solving methods, the research concentrated more on *knowledge representation*, rather than computer understanding. This new approach led to the development of specialized computer programs, called *expert systems*. Expert systems are designed for representing and handling knowledge in a narrow problem area, in contrast to the above-mentioned general problem solvers. Other differences between traditional AI and expert systems are [Hayes-Roth *et al.*, 1983]:

- Expert systems are able to perform difficult tasks at expert levels of performance.
- Expert systems provide explanations and justifications about their own inference process.
- The tasks of an expert system often classify into one of the following categories: interpretation, prediction, diagnosis, design, planning, monitoring, debugging, repair, instruction and control.

In the last years, the expert systems research has proved to be a successful one. Expert systems are nowadays frequently used in various fields, such as in medicine and in measurement and control. Besides expert systems there are also 'conventional' programs active in these fields, but there are a number of differences between these programs and expert systems [Jackson, 1990]:

- Conventional programs reason about the problem domain, whereas an expert system simulates human reasoning.
- In an expert system, the knowledge is separated from the reasoning strategy that utilizes the knowledge, whereas in a conventional program they are interlaced. The knowledge part is usually referred to as the *knowledge base* and the reasoning part is called the *inference engine*. The separation of knowledge and reasoning strategies makes it easier to maintain the system, because it is possible to modify each part separately. The knowledge in the knowledge base exists in different

forms, such as *production rules* (the most popular), *predicate logic*, *semantic nets* or *frames*. It depends on the nature of the problem and the experience of the designer, what knowledge representation is the most favorable.

- The knowledge in the knowledge base is usually some kind of *heuristic*. A heuristic is a 'rule of thumb'; it does not guarantee a solution but often gives an acceptable one. This is unlike an algorithm, that always reaches an (optimal) result.

2.2. Components of an expert system

Figure 2.2 shows a characteristic expert system that is decomposed into several modules [Chytil and Engelbrecht, 1987]:

- **User Interface**: Although often not recognized, the user interface plays a very important role in the various communication processes between the user and the expert system. Using various techniques, such as graphic displays, menus, icons, mouse operations and voice control, the user is able to establish a dialogue with the expert system. Important requirements of a user interface are uniformity and comprehensibility throughout all the dialogues. Because the operating speed of the expert system usually differs from the user's operating speed (especially when dealing with real time expert systems), the user interface is sometimes realized as a separate program or task that runs in parallel [Langlotz and Shortliffe, 1983].
- **Inference Engine**: This module is (besides the knowledge base) the basic part of an expert system. As mentioned in the previous paragraph, the inference engine is the part that performs the reasoning. It can employ various strategies, such as *state space searching* (solving problems by means of searching through state spaces), *pattern matching* (solving problems by means of identifying patterns) or *parsing* (solving problems by means of grammatical analysis).
- **Knowledge Base**: This module contains the knowledge about the problem domain and is utilized by the inference engine's reasoning strategies. Because this module is the core of an expert system, expert systems are also referred to as *knowledge based systems*. Figure 2.3 shows some examples of different knowledge representations in a knowledge base.

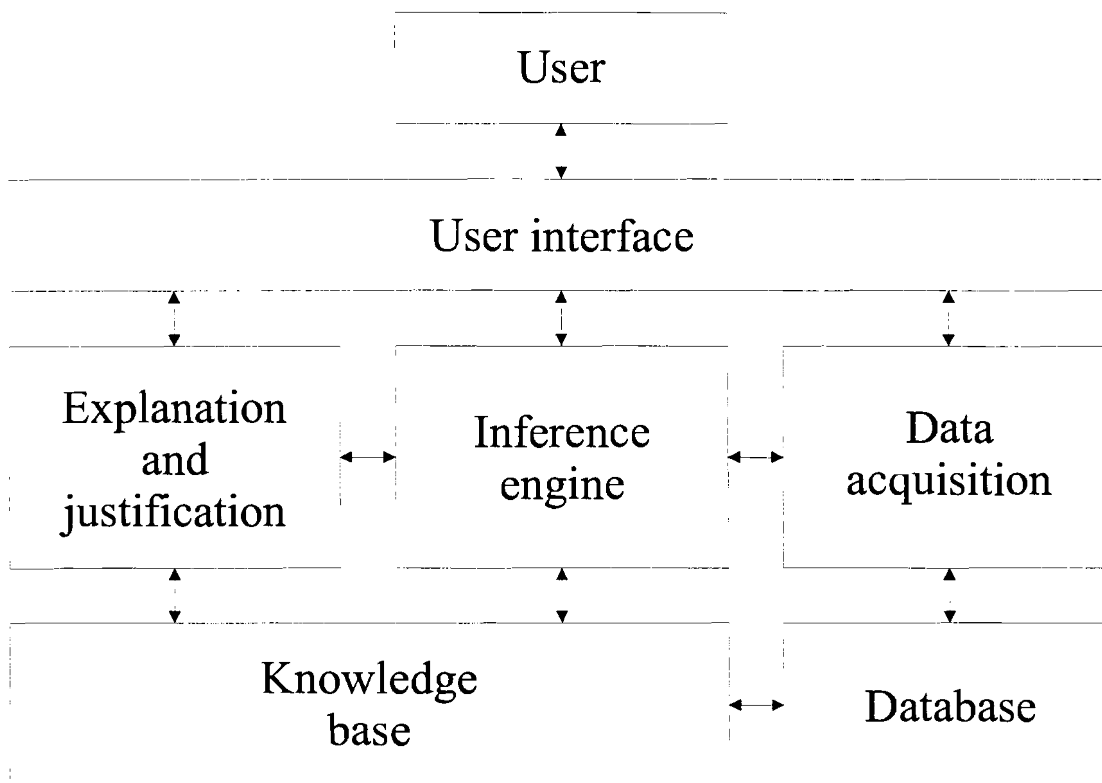

Figure 2.2: Components of an expert system

- **Explanation and justification**: The purpose of this module is to explain the inference engine's reasoning strategies to the user. The ability of clarifying the steps that led to the solution of a problem is one of the most important characteristics of an expert system. This, because the design of a knowledge base is a difficult process, especially the testing and debugging phases [Blom, 1990]. A common feature of the explanation or *debugging* module is the ability to reconstruct the inference chain after it has been completed. This makes it easier for the system developer to build, test and maintain the knowledge base.
- **Database**: The database normally contains two types of data:
  1. *Static data*: Facts and relations found in the problem domain. Examples are the age or weight of a patient.
  2. *Dynamic data*: Data, collected during a session. Examples are the patient's blood pressure or ECG.

  During or after a session, it is possible to transfer the dynamic data to the static data area for further use.

a) predicate logic

BROTHER(Paul, Eric) = Paul and Eric are brothers

b) rules

IF battery is empty THEN car will not start

c) frames

name	Peter
age	35
weight	83
height	185
occupation	student

d) semantic nets

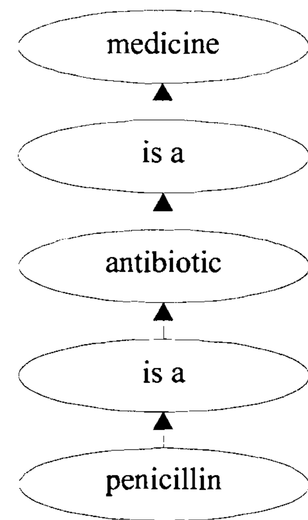

Figure 2.3: Various knowledge representations

- **Data acquisition**: The inference engine activates this module whenever it requires data. When activated, the data acquisition module tries to fetch the requested data. This implies gathering the missing data from a database or asking the user to supply the data if it is not available.

2.3. Production systems

As already stated, there are various ways to encode the knowledge in a knowledge base (figure 2.3), but the most popular are production rules. Expert systems that rely on production rules are usually referred to as *production systems.* A reason for their popularity is given by Newel and Simon, cited by [Blom, 1990]:

> *We confess to a strong premonition that the actual organization of human programs (i.e., in the human mind) closely resembles the production systems organization.*

and

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

2.3.1. Syntax

In a production system, a single rule has generally the following syntax:

$$P_1, \ldots, P_m \rightarrow Q_1, \ldots, Q_m ,$$

where $P_1, \ldots, P_m$ are called the *premises* or *conditions* and $Q_1, \ldots, Q_m$ are referred to as the *actions* or *conclusions*. Whenever a single rule is chosen (for example as a result of executing previous rules), the actions $Q_1, \ldots, Q_m$ are carried out only if the conditions $P_1, \ldots, P_m$ are satisfied (the rule then 'fires' or 'triggers'). These actions can cause other rules to fire, thus producing a chain reaction that is called the *inference chain*. This form of inferencing is called *forward chaining* or *data-driven* inferencing. Forward chaining is very resource demanding. Due to the 'avalanche-effect', chances are that in the long run (almost) every rule in the knowledge base will be checked.

Another form of inferencing is *backward chaining* or *goal-driven* inferencing. The backward chaining algorithm starts with one or more actions ('goals') and checks if all the conditions of the current rule are satisfied. If not, the algorithm searches for a rule that contains an action part that will satisfy the conditions of the current rule. When found, the (new) goal becomes to satisfy the condition part of the newly found rule, continuing until the answer is found or there is no rule available with a matching action part (indication that there is no answer to the problem). The majority of the expert systems use forward chaining as an inference mechanism, but there are also systems that are able to use both forward and backward chaining.

2.3.2. Communication between rules

Besides the *rule interpreter* (the part that decides when to select which rules), the inference engine also contains a module that holds the data, goal, statements and intermediate results that determine the current state of the problem [Jackson, 1990]. This module, called *working memory*, is the only means of communication between the various rules (figure 2.4).

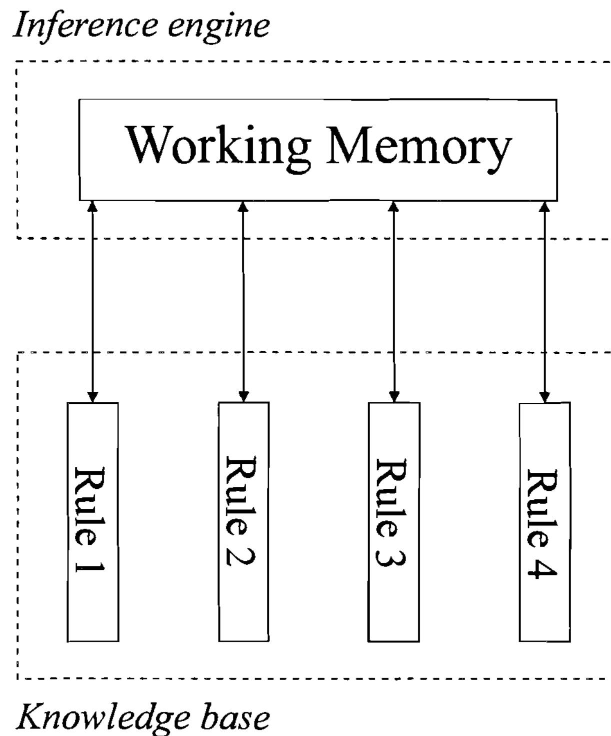

Figure 2.4: Communication between the various rules

Normally, a rule is not aware of the other rules in the knowledge base. All that a rule is capable of is modifying the data in the working memory en letting some other rule react to it. Using rules, it is not possible to encode *strategic knowledge* or *metaknowledge* (knowledge about the knowledge in the program) into the knowledge base, such as the implementation of a search strategy. For this purpose, some production systems expanded their syntax with so-called *metarules*. In contrast to 'normal' rules, metarules are able to direct the reasoning that is needed to solve a problem, rather than just perform that reasoning [Jackson, 1990].

It seems that the separation of (domain) knowledge and global strategy makes it easier to maintain or modify the knowledge base. However, adding or deleting rules sometimes produces unexpected side effects. This is because the rules in the knowledge base do not only express explicit knowledge but also contain some form of implicit knowledge (this is especially true when working with metarules). Therefore, modifying the knowledge base may not be as easy as it looks and should be done with great care. In order to simplify the maintenance of a knowledge base, production systems were developed that could decompose the knowledge into smaller parts

(called *chunks*). An example is the introduction of *goal hierarchies* [Blom, 1990]. This technique divides a goal into smaller goals (for example by using an AND structure), thus making the knowledge base more orderly.

2.4. Real time expert systems

The execution speed of an expert system is usually low, compared to conventional systems. A typical expert system spends 90 percent of its executing time searching through the knowledge base. Forward reasoning, for example, is an algorithm that is exponential time. The number of rules that will fire explodes exponentially with the depth of the inference tree. Also the backward reasoning algorithm is very resource demanding: every increment in the depth of the inference tree gives an exponential increase in the number of tree nodes and a combinatorial increase of the number of paths to search [Laffey *et al.*, 1988]. This phenomenon is called *combinatorial explosion* and has always been a controversial topic in AI research.

2.4.1. Expert systems operating in dynamic environments

According to [Laffey *et al.*, 1988], 'traditional' expert systems are not suitable for unstable and dynamic environments, such as the ICU:

> *Knowledge based systems operating in a real time situation (for example crisis intervention or threat recognition) will typically need to respond to a changing task environment involving an asynchronous flow of events and dynamically changing requirements with limitations on time, hardware and other resources.*

Humans that operate in these situations, tend to overlook valuable information, react too slowly or panicky when the flow of information becomes too great. The purpose of so-called *real time* expert systems is to 'reduce the cognitive load on users or to enable them to increase their productivity without the cognitive load on them increasing' [Turner, cited by Laffey *et al.*, 1988]. In contrast to 'normal' expert systems, real time expert systems must be able to recognize and respond to an external event within a certain period of time, called the *response time*.

There exist several definitions of *real time* in literature, but it is usually described as 'fast' or 'fast enough'. Other definitions are 'faster than a human can do it' or 'the system processes incoming data faster than it is arriving'. More formal definitions are found in [Blom, 1990].

2.4.2. The difference between traditional and real time expert systems

As stated above, real time expert systems operate in environments in which the data is not static. As a result, these systems will encounter new and complex problems [Laffey *et al.*, 1988]:

- **Nonmonotonicity**: During the execution of the program, the flow of incoming (sensor) data will not remain static.
- **Continuous operation**: A real time expert system must be robust. The discovery of a partial of complete failure of one or more parts of the expert system may not result into a system shutdown.
- **Asynchronous events**: Incoming events are often not scheduled, but occur randomly in time. Also, the events may differ in priority.
- **Interface to external environment**: Data is usually gathered from a set of sensors and not supplied from the user.
- **Uncertain or missing data**: A real time expert system must be able to detect uncertain, invalid or missing data.
- **High performance**: The system must respond rapidly to external events. This is one of the most important requirements of a real time expert system.
- **Temporal reasoning**: In real time environments, time is a very important variable. A real time expert system must be able to reason about past, present and future events.
- **Focus of attention**: When receiving an external event, the system must be able to focus its resources on the important goals, depending on the event.
- **Guaranteed response times**: The system must be able to respond by the time the response is needed.

- **Integration with procedural components**: It must be possible to integrate the real time expert system with 'conventional' software. This software will perform tasks such as I/O-operations or providing a user interface.

Although considerable effort is being put into developing real time systems, there still exist pressing problems that need to be solved:

- The system is not fast enough.
- The system has little or no capability for temporal reasoning.
- The system is difficult to integrate with conventional software.
- The system has little or no facilities to focus attention.
- There is no integration with a real time clock.
- There are no facilities for handling asynchronous inputs.
- The system is not equipped to handle software-hardware interrupts.
- The system can only efficiently obtain input from human sources.
- There are no methods for verifying and validating the shell or knowledge base.
- The system cannot guarantee response times.
- The system runs on hardware that was not built for harsh environments.

Considering the above-mentioned problems and limitations, the following features must be expected of a real time expert system:

- An efficient integration of numeric and symbolic computing.
- Continuous operation
- A focus-of-attention mechanism.
- An interrupt-handling facility.
- Optimal environment utilization; i.e., compiled instead of interpreted code.
- Predictability.
- A temporal reasoning facility.
- A truth maintenance facility.

2.5. Expert systems in medicine

During the late 1960s and early 1970s, the field of *Artificial Intelligence in Medicine* (AIM) arose. Expert systems, such as MYCIN (deals with infectious disease) and INTERNIST (models diagnostic reasoning performed by human clinicians) were developed in order to assist physicians with medical decision making [Miller, 1988].

Nowadays, medicine occupies a leading place among the numerous different kinds of expert systems now developed. Various prototypes of medical problem solvers have been designed in the last two decades. However, the regular use of expert systems operating in medicine is found only rarely, due to several reasons [Miller, 1988]:

- The human organism and human disease processes are very complex subjects.
- Currently, only a shallow knowledge exists on most diseases.
- Even constrained subspecialty areas of medicine require a huge amount of knowledge.
- There exists a lack of familiarity with computers on the part of many health care practitioners.
- Practicing physicians often have to deal with pressured time demands and legal implications.

2.5.1 Critiquing systems

Considering the above-mentioned complexity of medical knowledge and medical practice, it is very important to determine the best approach for an expert system to assist a physician. Traditionally, expert systems in medicine have attempted to *simulate* a physician's reasoning strategies. A typical expert system receives information about a patient and then produces a number of conclusions and recommendations, such as a list of possible diseases, a set of suggested tests or a treatment plan (figure 2.5**a**). Such a system 'has the clinical effect of trying to tell a physician what to do' [Miller, 1986]. A *critiquing system*, however, follows another approach (figure 2.5**b**).

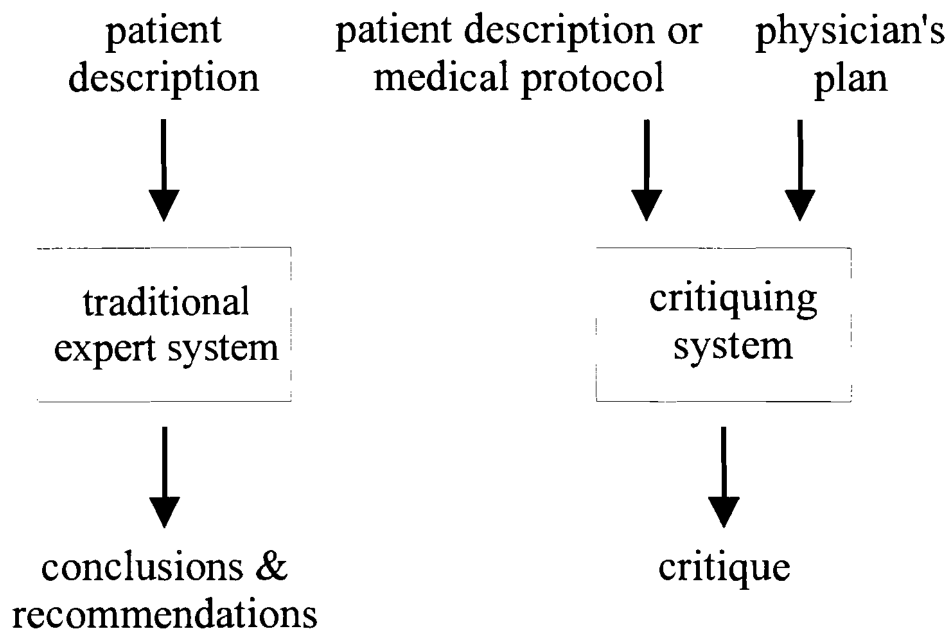

Figure 2.5: Comparison between a traditional expert system and a critiquing system, adapted from [Miller, 1986]

Besides the patient description, a critiquing system also accepts a physician's proposed plan or a medical protocol that must be carried out. The critiquing system then *critiques* the plan. In this way, it structures its advice around the physician's own thinking and style of practice. A critiquing system falls into the class of diagnostic expert systems.

2.5.2. Exploring the critiquing approach

According to [Van Der Lei, 1991]:

> *Certain workers in medical informatics have argued that, for some medical domains, critiquing the decisions of a physician is a preferred approach in providing decision support.*

The question why critiquing may be preferable in some medical domains can be approached from several perspectives [Miller, 1986]:

- **The clinical perspective**: The large amount of variation and subjectivity inherent in medical practice makes it almost impossible to try to tell a physician what to

do. Using the critiquing approach, the advice given to the physician can be presented in the form the physician can use best.

- **Critiquing as a form of practice-based education**: The critiquing approach allows the expert system to structure the detailed information around a physician's practice. As a result, critiquing is not only a form of decision support, but also a form of practice-based continuing medical education.
- **The computer science perspective**: Critiquing also contributes to the field of expert systems research. It is a way to structure explanation by an expert system of its internal reasoning processes.

Some domains are better suited for critiquing than other. Domains that suit the critiquing approach, possess the following characteristics [Miller, 1986]:

- There are a number of alternative choices.
- There are a number of different risks and benefits associated with the various choices in different patients.
- New treatment choices and new knowledge about existing features periodically alter the field.

An example of such a domain is the ICU, which is in practice a hybrid of various subspecialty areas of medicine.

3. The Intensive Care Information System (ICIS)

3.1. Patient Data Management Systems (PDM-Systems) in intensive care

An intensive care unit (ICU) attends patients that require continuous observation because their condition is often very unstable or even life-threatening. According to [Skillman, cited by Groen, 1995]:

> *An intensive care unit is a special care unit in a designated area of a hospital, which supplies specially trained medical, nursing and other personnel and diagnostic, monitoring and therapy equipment in order that seriously ill patients may be provided with close observation, intensive care, and with immediate recognition of potentially life-threatening complications, and prompt institution of indicated treatment.*

The first ICUs were constructed in the 1950 in the United States. At this time, ICUs were relatively simple units where there was no complex equipment available, such as heart monitors, dialysis machines or intelligent alarms. According to [Fairman, cited by Groen, 1995]: 'the nurses were the alarm'.

However, due to the development of new technology for diagnostic and therapeutic purposes, combined with the introduction of microprocessor technology, the amount of data collection in ICUs has increased enormously [Metnitz and Lenz, 1995]. Tasks, traditionally performed by nursing personnel are currently (partially) handled by computerized systems (table 3.1).

Due to these technological changes, the point has already been reached where the manual handling of these large amounts of data is very difficult to manage.

Function	Old Method	Currently practice
Monitoring	Direct observation by nurse	Continuous monitoring by machine and direct observation by nurse
Delivery of medication	Frequent interventions by nurse	Automated delivery systems and manual delivery
Ventilation	Flexible manual ventilation and inflexible mechanical ventilation	Flexible and versatile automated and manual ventilation
Blood pressure	Cuff and mercury gauge for on-off measurement	Direct and continuos measurement in the blood stream as well as non-invasive manual and automated measurements
Oxygen levels in blood	Blood sampled for analysis in laboratory	Continuous and non-invasive measurement of saturation, backed up by laboratory analyses.

Table 3.1: Technological changes in intensive care, adapted from [Groen, 1995]

3.1.1. The pros and cons of a PDMS

The purpose of a so-called *patient data management system* (PDMS), is to process the above-mentioned huge amount of data. According to [Metnitz and Lenz, 1995], these systems have their pros and cons:

- The collected data, combined with corresponding data (for example, data that is collected at different times or sites) is presented in the most appropriate form to the user in real time.
- The time, used for *charting* can be reduced up to 50 percent through computerized charting systems. However, factors such as the severity of the patient's illness and the amount of data documented through PDMS could result in increasing charting times.
- Human errors are common in handwritten documents, such as arithmetic errors or data omission. Computerized documentation, on the other hand is said to provide completely and readable documentation. However, research showed that the above-mentioned errors also exist in computerized documentation.
- The collected data may easily be processed for scientific analysis or quality control purposes. An example is the calculating of the so-called *patient scores*. The purpose of these scoring systems is to 'rate' a patient's disease and the amount of required patient care. For example, the APACHE (Acute Physiologic and Chronic Health Evaluation) and the SAPS (Simplified Acute Physiologic Score) scoring systems both express the 'amount of patient suffering'. Another example is TISS (Therapeutic Intervention Scoring System), that has been developed in order to determine the required workload.
- Medical computer systems are still expensive and the decision upon which system is to be purchased needs to be made carefully. Also, secondary considerations are necessary, such as 'what kind of peripheral equipment (ventilator, monitor) must be purchased' and 'is it possible to connect the PDMS to the existing equipment'. These decisions often are made by clinicians or managing directors that have no or little expertise in medical informatics.
- Sophisticated computer systems, such as PDM-Systems, need maintenance by specialized personnel, which may increase hospital costs.
- The implementation of a PDMS needs much time and resources and may decrease the productivity of an ICU. The starting time of a PDMS may be affected by first-time faults and is surely a burden for the staff. Also, if the PDMS fails to satisfy the customer's needs after implementation, it is very difficult to change to another one.

3.1.2. Classifications of PDM-Systems

PDM-Systems are classified into 3 categories [Metnitz and Lenz, 1995]:

- **Self-made-systems**: These systems are developed by informatic specialists that are working in the field of hospital communication as well as in the field of computing. Most of these systems are adapted to local needs and can hardly be transferred to other ICUs.
- **Minimal PDM-Systems**: Minimal PDM-Systems are able to collect information from various sources, such as a *Serial Distribution Network* (SDN) or laboratory. These systems, often PC-based, are usually easy to handle and cheap solutions for ICUs that cannot afford a large PDMS. ICIS classifies into this category.
- **Commercially available bedside-based PDM-Systems**: These are the most powerful PDM-Systems available today. The core of these systems is the information, obtained from the bedside. To process this information, a fast database response time is necessary, but often hard to achieve, especially when the amount of data grows constantly. As a result, they are often not enough to cover the demands of information management in an ICU. Further applications, such as bedside display of X-rays will need even more computer power than that provided by today's systems. Currently, there are just a few of these systems available in Europe, such as Atlantis, Carevue 9000, Chartmaster, Clinicomp, Clinisoft and Emtek. An overview of the current situation in Europe is presented by [Metnitz and Lenz, 1995].

3.2. ICIS

As stated above, ICIS (Intensive Care Information System) is a PDMS that falls into the category of the 'minimal PDM-Systems'. Started as a simple medicine program, ICIS has evolved into a complex database system that is now operating full-time in the ICU of the Catharina Hospital, situated in Eindhoven, the Netherlands.

ICIS has been developed in association with the medical personnel of the ICU. As a result, ICIS attempts to improve the communication between the various members of the medical staff. This philosophy differs considerably from the viewpoint of the

'bedside-based' PDM-Systems that are based on the information from the monitoring equipment. ICIS is implemented in Microsoft Access and has a patient-oriented user interface. **Note**: ICIS is developed in the Netherlands. Therefore, the language of the user interface is entirely written in Dutch.

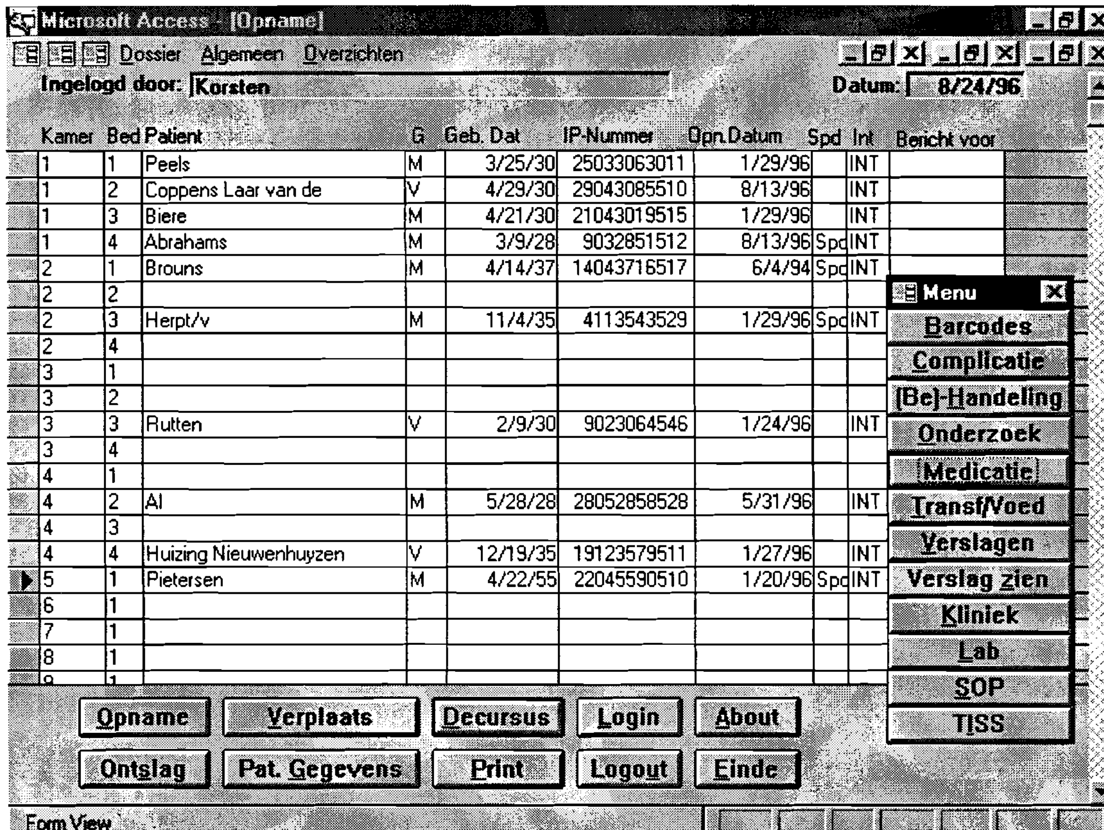

Figure 3.1: The main screen of ICIS

Figure 3.1 shows the main screen. It presents the *admission table*, containing information about the current administered patients, such as their room number, name, gender, identification number, admission date, etc. When the user has gained access to the system (by means of a login name and password), the user may request information about a patient, such as the patient's medication, physical examinations or complications. It is also possible to enter new information, such as adding a new medicine, the result of an examination or administering a new patient.

ICIS is divided into several components (visualized to the user by means of corresponding screens), all reachable from the main screen (figure 3.2):

- **Admission and admission diagnosis screen**: The purposes of the admission screen are:
    1. Administering a new patient.
    2. Adding or changing the data of an already administered patient, such as the patient's room number or allergies.

    When administered, the patient receives an *admission number* or *IC-number* that is admission-specific. This, in contrast to the so-called *patient number* or *IP-number* that is unique for every patient. When a patient is administered several times, the IP-number always remains the same, whereas the admission number changes with every admission.

    After entering the patient's data, the user is presented with the admission diagnosis screen. In this screen, the user may enter the patient's condition, active and non-active problems and medical history.

- **Discharge screen**: The user chooses the discharge screen whenever a patient is discharged from the ICU. Before discharging the patient, the user may enter discharge-specific data, such as the patient's condition or destination.
- **Patient information screen**: This screen presents an overview of the patient's data. The screen's content is for inspection only and is useful whenever a quick profile of the patient is desired.

Activating the Decursus button reveals a floating menu (figure 3.1) that provides access to the following screens (figure 3.2):

- **Overview medication and add medication screen**: The purpose of the overview medication screen is to present an overview of the patient's medications. By means of the add medication screen, it is also possible to add new ones.

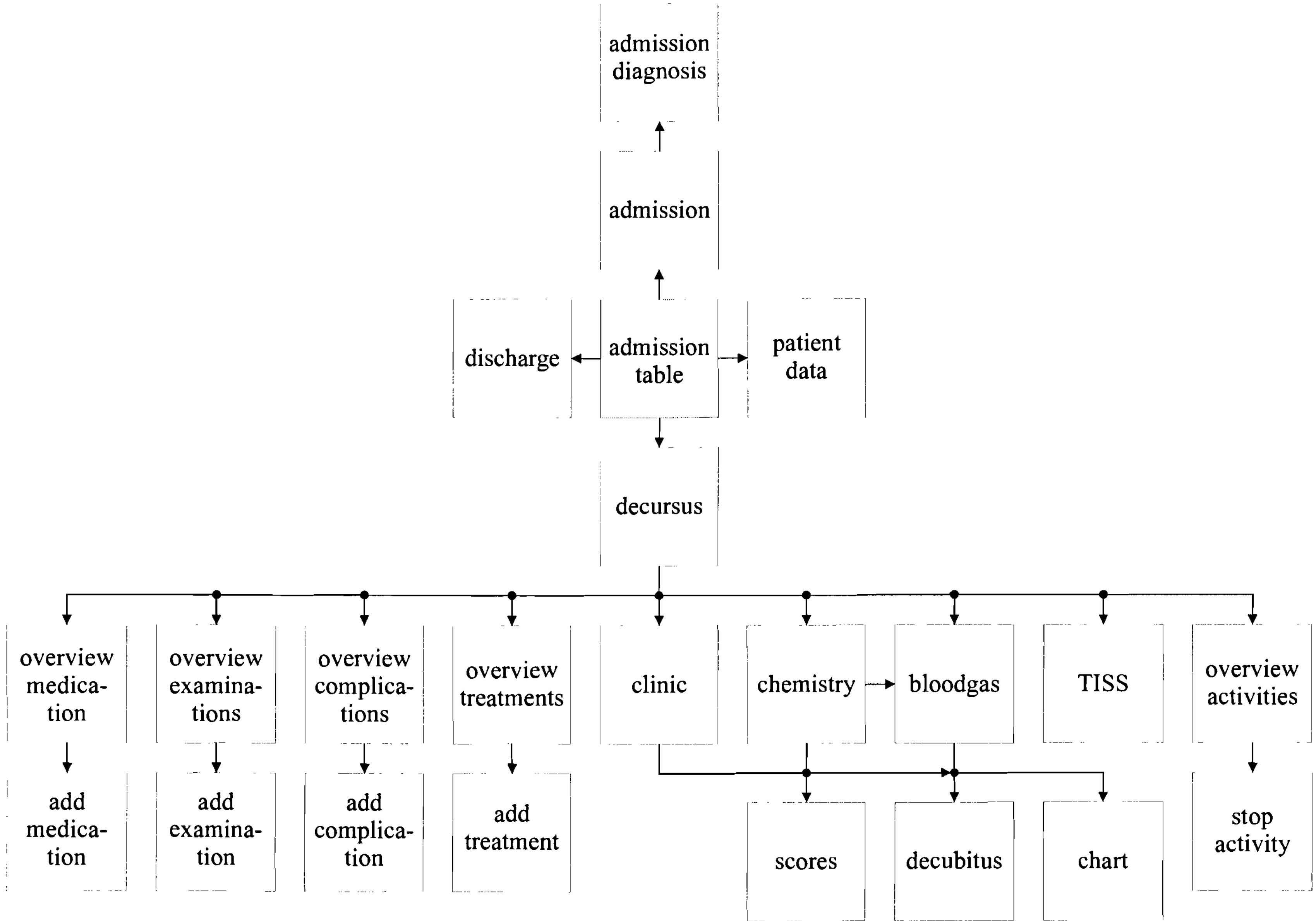

Figure 3.2: The various components of ICIS

- **Overview examinations and add examination screen**: The overview examinations screen shows the patient's current and previous medical examinations. Whenever the patient's condition demands a new examination, the user selects the necessary one by means of the add examination screen.
- **Overview complications and add complication screen**: The patient's complications are presented in the overview complications screen. Furthermore, the add complication screen enables the user to enter a new complication, along with the complication's category, starting date and suggested treatment.
- **Overview treatments and add treatment screen**: The purpose of this screen is to present an overview of the patient's treatments, including date, time and treating physician. New treatments are entered by means of the add treatment screen.
- **Clinic screen**: The purpose of the clinic screen is to enter and present the clinical variables and the Glasgow coma scale. These values are used to calculate the APACHE II and SAPS II scores.
- **Chemistry screen**: The screen enables the user to enter the results of the laboratory tests, such as the glucose, ASAT or ALAT values. Along with the clinical variables and the Glasgow coma scale, the APACHE II and SAPS II scores are calculated, using these values.
- **Bloodgas screen**: This screen shows the arterial and venous bloodgas values.
- **Scores screen**: ICIS is able to calculate several scores, such as APACHE II, SAPS II and decubites. The scores screen presents an overview of these scores.
- **Decubites screen**: This screen presents a detailed overview of the decubites score.
- **Chart screen**: This screen presents a graphical overview of several values, taken from the clinic, chemistry and bloodgas components.
- **TISS screen**: The TISS screen displays a detailed overview of the patient's TISS score.
- **Overview activities and stop activity screen**: The overview activities screen displays the activities of a selected patient that are to be carried out by the medical staff, such as weighing the patient or turning the patient around. These activities may be connected with other decursus items, such as the patient's medications or examinations items. In the stop activity screen, the user may (temporarily) halt an

activity. As a result, this activity will no longer be visible on the so-called *activity list*.

In ICIS, each component is linked to a corresponding Microsoft Access table (for example, table 1.1 shows a part of the admission table). All the actual data is stored in these tables and is accessible by means of SQL (Structured Query Language) statements.

3.3. The monitoring equipment

An overview of the ICU's monitoring equipment is shown is figure 3.3.

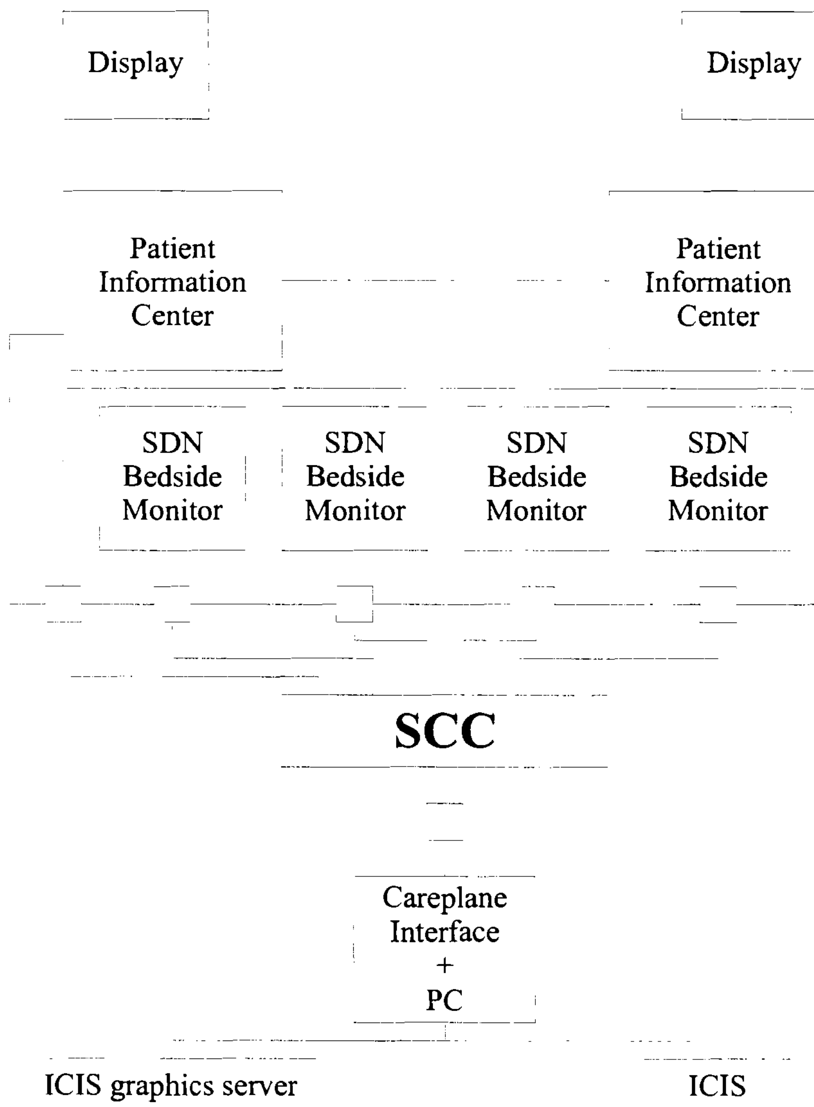

Figure 3.3: The monitoring equipment

3.3.1. The SDN Monitoring network

The Hewlett-Packard SDN (Serial Distribution Network) is a local area communications network designed to share patient data among bedside monitors and other information systems that are connected to the network. The SDN allows real-time transfer of digitized patient data between the following components:

- **The System Communications Controller (SCC)**: This component is the core of the SDN. It collects the patient data and synchronizes all the instruments.
- **Bedside monitors**: The purpose of the bedside monitors is to measure and transmit the patient's data onto the SDN. There are currently 22 bedside monitors operating at the ICU, each of them capable of measuring the following physiologic variables:

  - HR (Heart Rate)
  - ABP (Arterial Blood Pressure)
  - PAP (Pulmonary Arterial Pressure)
  - CVP (Central Venous Pressure)
  - S_pO_2 (Oxygen saturation)
  - RESP (Respiration frequency)
  - NBP (Non Invasive Blood Pressure)
  - PULSE (Pulse frequency)
  - CO (Cardiac Output)
  - BLOODT (Blood Temperature)
  - T_1, T_2, T_2-T_1 (various temperatures)

- **Patient information centers**: The patient information centers display the patient's physiologic variables monitoring data graphically to the user, by means of a display or terminal. At present, there are 4 patient information centers and displays connected to the SDN.

After collecting, the SCC sends the monitoring data to the *careplane* interface, which forms a bridge between the SDN and a personal computer. The purpose of this

computer is to filter the data, after which it is send to the main network of the ICU. In this network, the filtered data is collected and processed by other programs, such as ICIS or the *ICIS Graphics server*.

3.3.2 The ICIS graphics server

In order to display the (filtered) data in ICIS, the ICIS graphics server has been developed which is capable of displaying the patient's physiologic variables (figure 3.4).

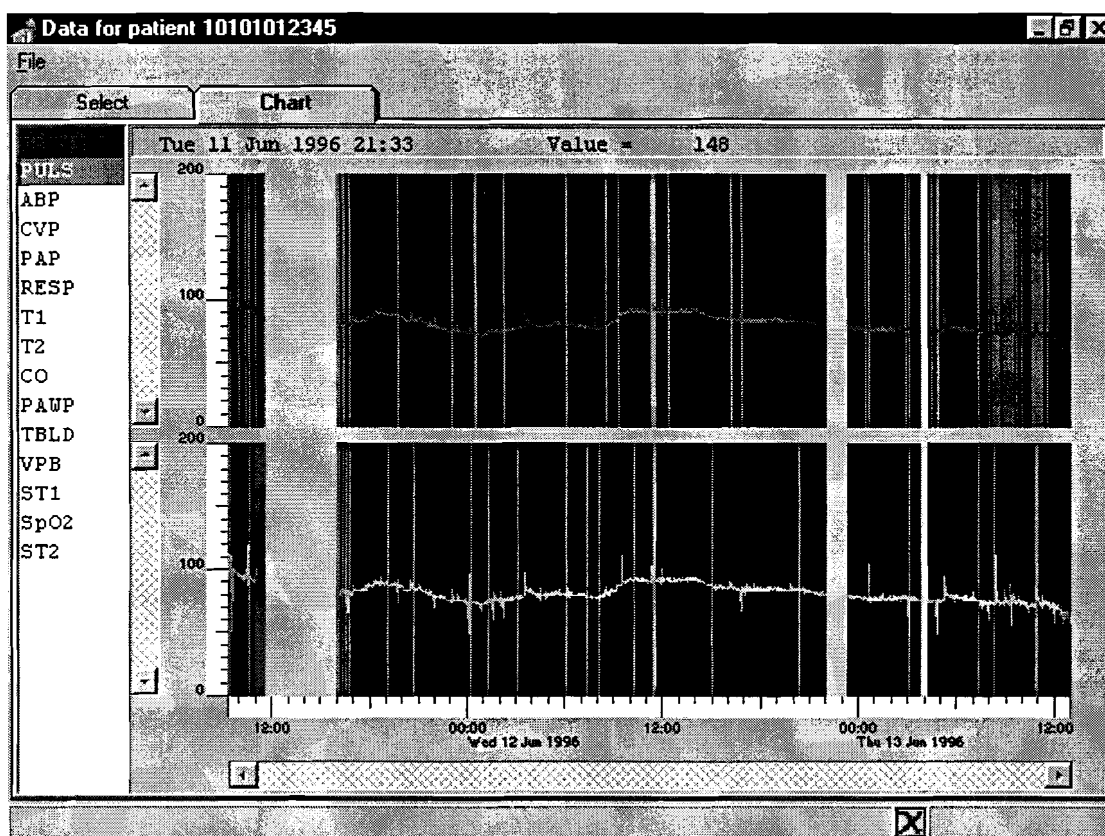

Figure 3.4: The ICIS graphics Server

3.3.3. Current stage in development

At present, the careplane interface is still in the development phase. As a result, it is not possible to connect ICIS directly to the SDN or to the ICIS graphics server. The physiologic variables are still entered manually into ICIS by the medical staff, using the patient information centers. However, in the near future, the SDN network and the ICIS graphics server will both become an integrated part of ICIS.

4. Development tools and techniques

4.1. Developing clinical systems using rapid prototyping

Software development is sometimes more considered an art than a science. However, in the last two decades, various attempts were made to formalize the process of developing correct commercial systems. This process, called *software engineering*, usually consists of five stages [Wyatt, 1994]:

- Capture the user's needs on paper.
- Write a detailed specification of the system's components and functions.
- Transform the specification into a program.
- Test the system and train operators.
- Maintain the system (correct 'bugs') and add new features.

However, this approach is often not a successful one. Research showed that a large percentage of today's commercial systems 'simply don't work very well' [Andriole, 1992]. This conclusion is also true of many clinical systems, due to the following reasons [Wyatt, 1994]:

- Clinical tasks and their environment constantly evolve. According to a developer: 'detailed plans prepared in advance are likely to be obsolete by the time the programs are implemented'.
- Physicians often act upon incomplete and uncertain information. They need a system that is easy to use: terminals must be ubiquitous, system response must be immediate, necessary data should always be on-line, accessible and confidential, and very little training should be required.
- It is not unusual to find poor communication or even distrust between physicians and computer professionals.
- There often exists a lack of predictability. When operating a clinical system, physicians look for predictability especially in the way their commands control the system.

- Clinical problems are usually viewed from the *information technology* perspective, although in practice they often require a mixture of solutions. This may result in a failure of evaluating the system's impact on clinical process and outcome.

4.1.1. Evolutionary rapid prototyping

In order to develop systems that possess the above-mentioned characteristics, a more flexible and dynamic developing strategy was created, called the *evolutionary rapid prototyping* approach (figure 4.1).

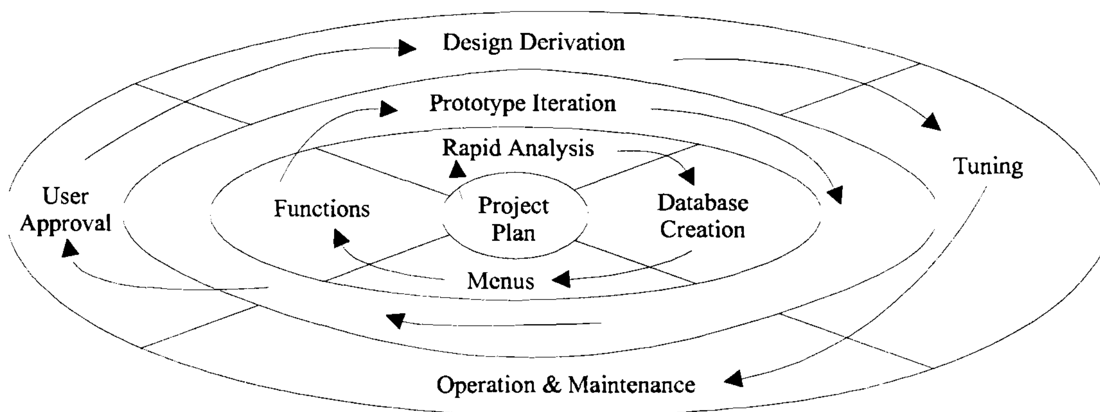

Figure 4.1. The evolutionary rapid prototyping process, adapted from [Connell and Shafer, 1989]

A model that is developed by means of the evolutionary rapid prototyping approach is called an *evolvable prototype*. Two formal definitions of such a model are presented in [Connell and Shafer, 1989]:

> *An easily modifiable and extensible working model of a proposed system, not necessarily representative of a complex system, which provides users of the application with a physical representation of key parts of the system before implementation.*

and

> *An easily built, readily modifiable, ultimately extensible, partially specified working model of the primary aspects of a proposed system.*

4.1.2. The path from prototype to application

After analyzing and specifying the user's basic needs, the system developer creates an initial prototype, sometimes called a *demo*. It is often nothing more than a skeleton and its purpose is to give the user a basic idea of the system's user interface.

The next stage of the rapid prototyping strategy is the *prototype iteration* phase. During this continuous iterative process, the prototype evolves with each iteration. First, the user may suggest some modifications or enhancements, after which they are implemented in a new version of the prototype (figure 4.2)

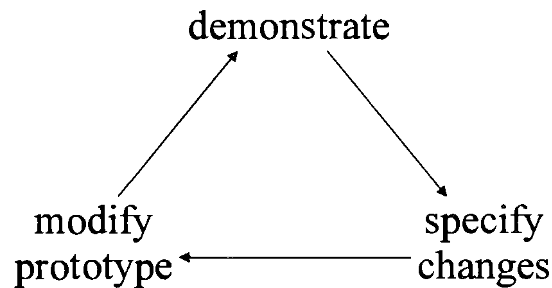

Figure 4.2: The prototype iteration process

This process continues until the user is completely satisfied with the functionality embodied in the refined prototype. According to [Connell and Shafer, 1989]:

> *This is a milestone that indicates that prototype iteration is over. Working together, prototypers and users have discovered what information the system must capture and produce to make it optimally useful ... In other words, prototype iteration is over when the user says, 'This version of the prototype does everything I would like the system to do and does it just the way I would like things done'.*

However, this does not mean that the system is ready for delivery. Usually, the system is not adequately documented, the system's response is too slow and it is unknown how it will perform under future loads of data and user volumes. In order to transform the system prototype into a final application, two more steps must be taken:

1. **Design derivation**: This step may include several tasks, such as encoding the system in some low-level language, writing the final documentation and training application users.
2. **Performance Tuning**: This includes tasks such as stressing the prototype (determining and documenting the system's bounds, for example by increasing the number of users), optimizing (and minimizing) the data structures and modifying the configuration (for example upgrading the hardware).

The rapid prototyping approach is an excellent strategy for developing dynamic applications, such as the above-mentioned clinical systems. However, sometimes rapid prototyping turns into an other 'software developing strategy', called *quick and dirty prototyping.* The term quick and dirty describes the approach of quickly bringing up a version of the system and then modifying it until the customer can grant minimal approval [Connell and Shafer, 1989]. The main difference between quick and dirty prototypes and evolvable prototypes is that changes in evolvable prototypes are usually temporarily, while in quick and dirty prototypes they are (mostly unintended) often permanently. The resulting software is almost sure to be expensive to maintain because the code has been patched many times, even before it is delivered.
A system developer that uses the rapid prototyping approach, must always keep in mind that, although quick and dirty prototyping seems more beneficial at the start (mainly because a working version will become rapidly available), further development and maintenance become more difficult with each new prototype upgrade.

4.1.3. Developing a critiquing system by means of rapid prototyping

In the last decades, a number of artificial intelligence programs were developed, using rapid prototyping. The philosophy here is that is not possible to specify requirements for AI applications, since there is no way of knowing what can be accomplished until something is tried. This philosophy is not always without risk because it may easily lead to quick and dirty prototypes. However, with a well-devised project plan and thorough analysis, it is possible to develop well-structured AI applications [Connell and Shafer, 1989].

Besides analysis, an other important consideration when developing a evolvable prototype is what kind of *tools* to use. As a development technique, rapid prototyping is unique because it is very dependent on specific kinds of hardware and software [Connell and Shafer, 1989]. The most common tools, used in rapid prototyping, are a *Rapid Application Development* (RAD) tool and a *DataBase Management System* (DBMS), but other tools are also used, depending on the nature of the application. For example, a critiquing system must contain an inference engine, a knowledge base and a debugger.

This paper describes the development of CritICIS. CritICIS is a critiquing system that is developed by means of the evolutionary rapid prototyping technique as well as a set of tools, consisting primarily of two elements:

- **Borland Delphi**: A visual programming environment, especially designed for RAD (Rapid Application Development), particularly applications using database access.
- **The SIMPLEXYS toolbox**: A collection of tools to design real time expert systems. The toolbox contains an expert system programming language, a rule compiler plus extensions, an inference engine and a Tracer/Debugger.

4.2. The Borland Delphi programming environment

Delphi is a graphical RAD tool, designed to build Microsoft windows applications. Delphi includes an *object-oriented* (OO) Pascal compiler and debugger, a visual design environment and strong database tools. Borland designed Delphi for developing prototypes and converting them into commercial products.

4.2.1. Component technology

Traditionally, software products were mainly composed of large amounts of code. As a result, developing a commercial product usually required a considerable amount of time and resources, especially maintaining the program or reusing program code. However, in the last decade, the field of software development has shifted its focus from 'traditional' programming to *Object Oriented Programming* (OOP). Software,

developed by means of OOP, no longer contains large blocks of code, but is constructed of various *objects* that communicate with one another. Every object has a clearly specified purpose, such as a list of records or a text file.

The two most important programming tools for writing Object-Oriented (OO) programs are *encapsulation* and *inheritance*. An object or class usually *encapsulates* all its data and functions, dividing them into two categories:

- **Private data**: The data and functions in this category are hidden from the outside world. As a result, they are also inaccessible outside the class.
- **Public data**: The data and functions in this category are visible outside the class and are the only means of communication between the class and the outside world. Public data and functions are also referred to as *member data* and *member functions*.

The main advantage of encapsulation is that the exchange of information between the class and its environment (usually consisting of other classes) takes place via a well-structured interface. To obtain information about a class, the programmer just calls the (visible) member functions, without paying attention to the internal functioning of the class. As a result, encapsulation simplifies the maintenance and reusability of a class.

The second important OOP technique is called inheritance. To add new operations and data to an existing class, the programmer derives a new class from the existing one and inserts the needed data and functions. It is also possible to override existing member functions, either to replace them or enhance their functioning. Using inheritance, programmers are able to extent the functioning of an existing class, without the necessity to develop everything from scratch.

The continuation of OOP is called *component technology*. The main difference between OOP and component technology is that OOP software consists of one program that is divided into various objects, whereas software that is developed by means of component technology is composed of various modules, called *components*.

Component technology is a superset of OOP: components are extended objects and present various levels of complexity. For example, a component may hold an 'ordinary' object, but it may also contain a word processor, a spreadsheet or a video-player. Components communicate by means of various protocols, such as *Dynamic Data Exchange* (DDE) *or Object Linking and Embedding* (OLE).

4.2.2. Visual development environments

A special type of components is the so-called *visual element.* Visual elements are graphical components, such as windows, menus or buttons. Using visual elements, a system developer is able to rapidly construct the program's user interface, serving as an initial prototype (skeleton) for further development. During the prototype iteration phase, the developer implements additional code and the prototype will gain functionality with each iteration. Development environments that contain visual elements are referred to as *visual development environments.*

4.2.3. Building applications by means of Delphi

As stated above, Delphi is a visual development environment. A Delphi application consists of so-called *forms* that each may contain a set of components (figure 4.3).

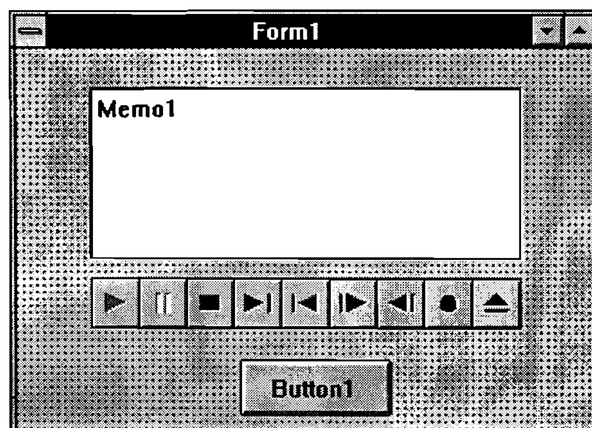

Figure 4.3: A simple Delphi application

Figure 4.3 shows a simple Delphi application, consisting of a form and three components: a memo field that provides an area for text manipulation, a video player and a button. In Delphi, each component generally holds three attributes:

- **Properties**: A property definition declares a named attribute for the component, such as the caption of a form, the size of a font or the name of a database.
- **Events**: Events are connections between user actions (for example a mouse click) and the code, written by the component developer.
- **Methods**: Methods are procedures or functions built into a component and their purpose is to direct a component to perform a specific action or return a certain value not covered by a property.

When comparing objects to components, properties are comparable to member data and methods to member functions.

Delphi's components are all part of an object hierarchy, called the *Visual Component Library* (VCL). Besides using Delphi's built-in components, it is also possible to develop new ones or use third-party components.

4.3. The SIMPLEXYS toolbox

SIMPLEXYS is an expert system toolbox, designed for developing real time expert systems. In order to accomplish fast response times, SIMPLEXYS was implemented in the efficient programming language Pascal. This, in contrast to the majority of today's expert systems that are usually implemented in LISP (LISt Processing), which is 'the main programming language of artificial intelligence' [Jackson, 1990]. However, compared to Pascal, LISP programs are usually very slow and are therefore not suited for real time applications [Blom, 1990].

4.3.1. The SIMPLEXYS programming language

The programming language is the core of the SIMPLEXYS toolbox and was originally developed to formulate and solve problems in the domain of patient monitoring and clinical control systems. The SIMPLEXYS programming language is rule based: the knowledge in the knowledge base is encoded into production rules. As a result, expert systems that are build with SIMPLEXYS are so-called production systems, similar to the ones described in section 2.3.1. However, SIMPLEXYS rules are not realized as *implications*, but implemented as *assignments*. For example, the

production rule **`if A and B then C`** is most approximately translated into the SIMPLEXYS format **`C := A and B`**.

SIMPLEXYS is based on a three-valued logic. Besides the logical values TR (true) and FA (false), a SIMPLEXYS rule also may have the value PO (possible). According to [Blom, 1990]: 'Three-valued is a better approximation of human reasoning, because in many practical situations it will neither be possible to decide that a proposition is true nor false'. There have been other attempts to represent uncertainty in human reasoning, such as Fuzzy Logic or Certainty Factors. However, these methods were not implemented in SIMPLEXYS, in order to increase efficiency.

Generally, a rule in SIMPLEXYS consists of two to four parts:

- rule header
- rule body
- initial condition (optional)
- THELSEs (optional)

1. **The rule header**: The rule header contains the rule's symbolic name, followed by a description of the rule:

```
EMPTY_BATTERY: 'The car's battery is empty' {rule header}
BTEST Voltage < minVoltage {rule body}
```

Rule 4.1: A simple SIMPLEXYS rule

The above rule checks whether the car's battery is empty, by means of measuring the battery's voltage.

2. **The rule body**: The rule's body is the second line in a rule and indicates the *rule's type*. The SIMPLEXYS programming language distinguishes two types:

- **Primitive rules**: These rules are independent of other rules. The only means of communication between primitive rules is to influence the working memory (section 2.3.2). In SIMPLEXYS, primitive rules obtain their value by direct assignment. There exist 5 different primitive rules:

  1. FACT rules. The value of a FACT rule is a constant value (TR, PO of FA).
  2. ASK rules: These rules obtain their value through communication with the user, by means of the keyboard.
  3. TEST rule: The value of a TEST rule is determined by means of one or more Pascal statements. A special TEST rule is the so-called BTEST rule, which obtains its value by means of a Pascal *Boolean expression.* An example of such a BTEST rule is shows in rule 4.1. In this rule, `Voltage` is a Pascal function that returns the battery's current voltage.
  4. Memo rules: These rules are often used for memory purposes.
  5. State rules: State rules specify the current *context.* They are assigned either as an initial conclusion or via the protocol: a set of ON statements that describe so-called *context switches* (see also section 4.3.2).

- **Evaluation rules**: In contrast to primitive rules, evaluation rules are dependent of other rules, either primitive or other evaluation rules:

```
FAULTY BATTERY: 'The car's battery is faulty'
EMPTY_BATTERY or LOOSE_WIRE
```

Rule 4.2: A SIMPLEXYS evaluation rule

The purpose of these rules is to direct the reasoning, required to solve a problem, similar to the strategic rules, described in section 2.3.2.

3. **Initial condition**: The purpose of this (optional) section is to initialize the rule's value (either TR, PO or FA):

```
LOOSE_WIRE: 'One of the battery's wires is loose'
BTEST CheckForLooseWires
INITIALLY FA
```

Rule 4.3: A primitive rule that is initially FALSE

4. **THELSEs**: By means of THELSEs, it is possible to add consequences to a single rule. There exist three types of THELSEs: THENs , ELSEs and IFPOs. The THEN part is executed whenever the rule's value becomes TR, the ELSE part when the rule's value becomes FA and the IFPO whenever the rule's value becomes PO.

```
LOOSE_WIRE: 'One of the battery's wires is loose'
BTEST CheckForLooseWires
INITIALLY FA
THEN GOAL: FASTEN_LOOSE_WIRE
```

Rule 4.4: Rule 4.3, expanded with a THELSE (THEN)

Whenever the result of the TEST rule becomes TR (i.e. one of the wires is loose), the THEN part is carried out, firing another rule that fastens the loose wire. Besides firing other rules, a THELSE can also be followed by a value (TR, PO, FA), directly setting the value of another rule, or by a DO 'Pascal procedure', executing some Pascal procedure or function.

The SIMPLEXYS language enables the programmer to perform various logic operations with the rules. These operations include the familiar Boolean logic operators, such as the *monadic* `not` operator and the *dyadic* `and` and `or` operators (see for example the `or` operator in rule 4.2). However, besides the Boolean values TR en FA, SIMPLEXYS also uses PO, representing 'possible' or 'unknown'. As a result, the SIMPLEXYS programming language contains two additional monadic operators, namely `must`, meaning 'guaranteed to be true' and `poss`, which means 'no definite value can be determined'. Besides these monadic operators, SIMPLEXYS also introduces the new dyadic operators `ucand` ('unconditional `and`'), `ucor` ('unconditional `or`') and `alt` ('logically equivalent alternative').

4.3.2. The SIMPLEXYS rule compiler and extensions

In a SIMPLEXYS application, the knowledge base consists up to 7 sections. The first 5 sections are optional and contain Pascal code, necessary when one or more rules need an interface with the outside world (for example, see rules 4.1 and 4.3).
The program sections and their corresponding *keywords* need to appear in the following order:

1. **DECL**: Declarations (optional)
2. **INITG**: Global initializations (optional)
3. **INITR**: Run initializations (optional)
4. **EXITR**: Run exit initializations (optional)
5. **EXITG**: Global exit code (optional)
6. **RULES**: The rules
7. **PROCESS**: The protocol

The first section, called declarations, contains the types and variables that are used by initializations, exit code, TEST rules and THELSE DOs. Program sections two through five contain Pascal *code*, which is included into the SIMPLEXYS inference engine by the rule compiler. The statements in the INITG section are executed when the expert system starts up, whereas the EXITG section contains code that will be executed whenever the system shuts down. In contrast to the INITG section, the code in the INITR section is executed at the beginning of every run. Similarly, the EXITG section is called at the end of each run. The concept of internal runs is explained in section 4.3.3.

The RULES section contains the actual rules, such as the rules 4.1 through 4.4, whereas the last section (PROCESS) contains the expert system's dynamics. A typical SIMPLEXYS application consists of various *contexts* or *states*, determining the system's current state. Each state holds a set of goals, only relevant in that particular state. By means of so-called *trigger rules* or *state transitions*, the system is able to change its active states. The PROCESS section contains all these state transitions, also referred to as ON statements, because of their syntax:

ON ***Trigger*** **FROM** ***FromList*** **TO** ***ToList***

Whenever the ***Trigger*** evaluates to TR and if all states in the ***FromList*** are active, the transaction takes place, resulting in the activation of all the states in the ***ToList***. This process ends if there are no more active states left, resulting in the shutdown of the expert system.

Once the knowledge base has been created, it is processed by the so-called *Rule Compiler*. This tool, implemented as a separate program, compiles the rules into an internal format, used by the inference engine. In contrast to 'conventional', expert systems, all the searching takes place by the rule compiler during the compilation of the knowledge base. As a result, the algorithm, employed by the inference engine is *linear with time*. Expert systems, build with the SIMPLEXYS toolbox can be executed at high speed, which is an essential aspect of real time expert systems.

The conversion process from knowledge base to expert system consists of various steps, shown in figure 4.4.

First, the rule compiler translates the knowledge base into an internal format, saved in six files. Each file contains a specific part of the knowledge base:

- **rinfo.qqq**: All the arrays and tables used for representing the rules and their mutual connectivity
- **ruses.qqq**: The Turbo Pascal units, used by the knowledge base
- **rtest.qqq**: The sections, defined in the TEST rules
- **rdodo.qqq**: The collection of the DO sections
- **rinex.qqq**: The initialization and exit sections
- **rhist.qqq**: Information about the history sections

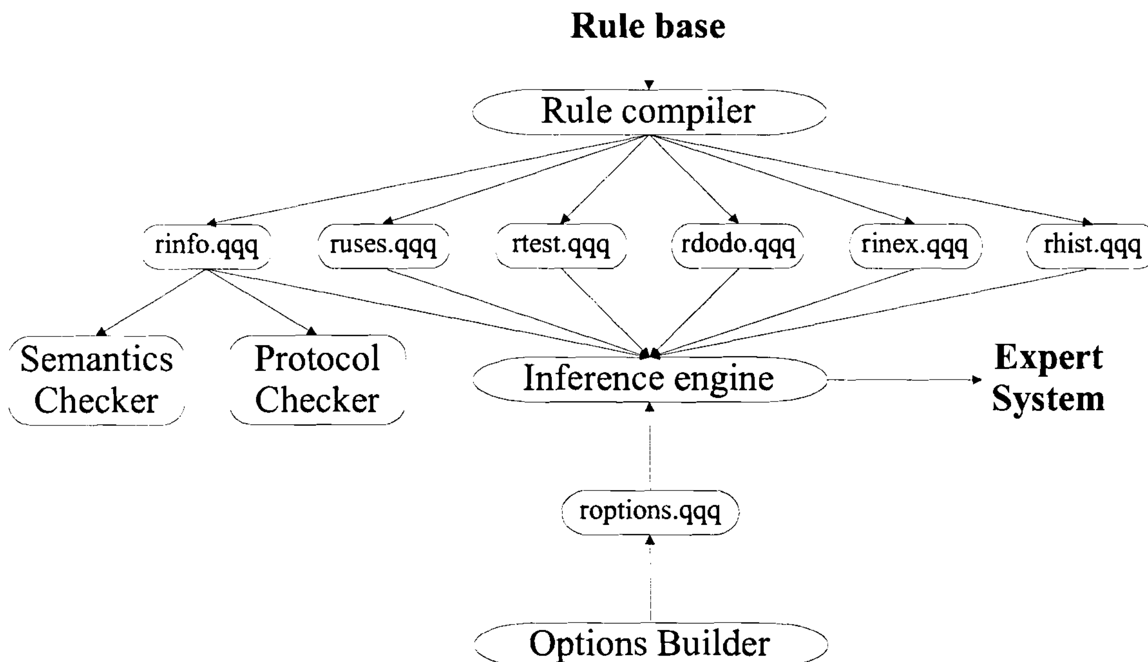

Figure 4.4: The path from knowledge base to expert system

During the translation process, the rule compiler checks for some syntax errors and simple semantic errors. In order to perform a more thorough error checking, two programs were designed to check the knowledge base for various errors, both using the rinfo.qqq file as input:

- **The semantics checker**: This program checks the knowledge base for several semantic errors
- **The protocol checker**: The protocol checker is used for the detection of errors in the PROCESS section, including syntax, topology and dynamic errors

Another tool is the *options builder*, that provides a way for the user to select various run-time options, such as debugging options.

The last step in the process is to combine all the various '.qqq' files with the SIMPLEXYS inference engine, resulting in an executable expert system. The combining process is carried out by the Turbo Pascal compiler, which automatically checks the Pascal sections in the knowledge base for syntax errors.

4.3.3. The SIMPLEXYS inference engine

As stated in section 2.2, the inference engine is the part of the expert system that performs the reasoning. In a typical expert system, the data changes over time, making it necessary to evaluate the various rules several times, perhaps including different goals. In SIMPLEXYS, every evaluation of the goals is called a *run* and the inferencing performed in one run is called a *single run inference*. During one run, all primary goals of the current context(s) are evaluated and the next context is determined, by means of the following steps:

- Update the time.
- Execute the run initialization code (INITR procedure)
- Execute the matching THELSEs of all FACT, MEMO and finally those of all STATE rules
- Try to perform a context switch
- Execute the run exit code (EXITR procedure)
- Undefine all TASK, TEST and EVAL rules for the next run

The inference process of a SIMPLEXYS expert system consists of a sequence of single runs, called the *global inference process*. This process consists of the following steps:

- Initialize the conclusions and history values of all rules
- Obtain the conclusions of those FACT rules, which did not obtain a value by means of the INITIALLY keyword
- Initialize the time
- *repeat*
  do a single run inference
  until all stated are inactive
- Execute the global exit code (EXITG procedure)

Whenever the expert system has finished, it is possible to examine its reasoning strategies. This is one of the most important features of an expert system: the ability to

provide explanation and justification about its internal reasoning strategies. In SIMPLEXYS, this is accomplished by means of the *Tracer/Debugger*. With this tool, it is possible to examine the inference engine strategies, after as well as during the inference process.

Production systems employ both *forward reasoning* and *backward reasoning*. Generally, by means of the structure of the rules (using assignments instead of implications), SIMPLEXYS is backward or goal oriented. This, in contrast to other expert systems that usually employ forward reasoning. However, by means of THELSEs, it is also possible to perform forward reasoning (for example rule 4.4). Finally, the determination of context switches (by means of evaluating the trigger rules) again is a form of backward chaining, making the total inference process rather complex, but also suitable for solving various classes of problems.

4.3.4. Adapting the SIMPLEXYS inference engine to object-oriented environments

In order to adapt the SIMPLEXYS inference engine to the above-mentioned object-oriented environments, it was implemented as a Delphi component (called *TInfer*, where the capital T stands for 'Type'). To accomplish this, the global inference process is broken down into a set of methods and properties, consisting of:

- **StartUp**: Performing the necessary initialization tasks
- **FirstRun**: Performs the inference engine's first single run
- **NextRun**: Performs the next run (if any)
- **CloseDown**: Performs the necessary exit code
- **Enabled**: Indicates of the SIMPLEXYS inference engine is still active

A program that utilizes the `TInfer` class now calls the various methods until the inference engine has completed its task (listing 4.1)

```
procedure GlobalInference
var
  Infer: TInfer; {SIMPLEXYS inference engine class}
begin
  {initialize inference engine}
  Infer.StartUp;

  {perform single runs until the inference engine is ready}
  if Infer.FirstRun <> ready then
    while Infer.NextRun <> ready do;

  {Execute exit code}
  Infer.CloseDown
end;
```

Listing 4.1.: Global inferencing, using the TInfer Class

As shown in listing 4.1, the methods **FirstRun** and **NextRun** return a code (ready or busy), indicating if the current run was also the last run. The purpose of the **Enabled** property is to check (usually in multitasking environments) whether the inference engine is still active.

Besides the above-mentioned member functions, the **TInfer** class also contains other member functions, used for handling the data in the two buffers, and various private data structures and functions that are called internally by the component's methods.

5. Critiquing ICIS: The development of CritICIS

5.1. An introductory example

The main purpose of CritICIS is to provide decision support to the users of ICIS. In order to establish a straightforward communication with the users of ICIS, the user interface of CritICIS is integrated into the user interface of ICIS (figure 5.1).

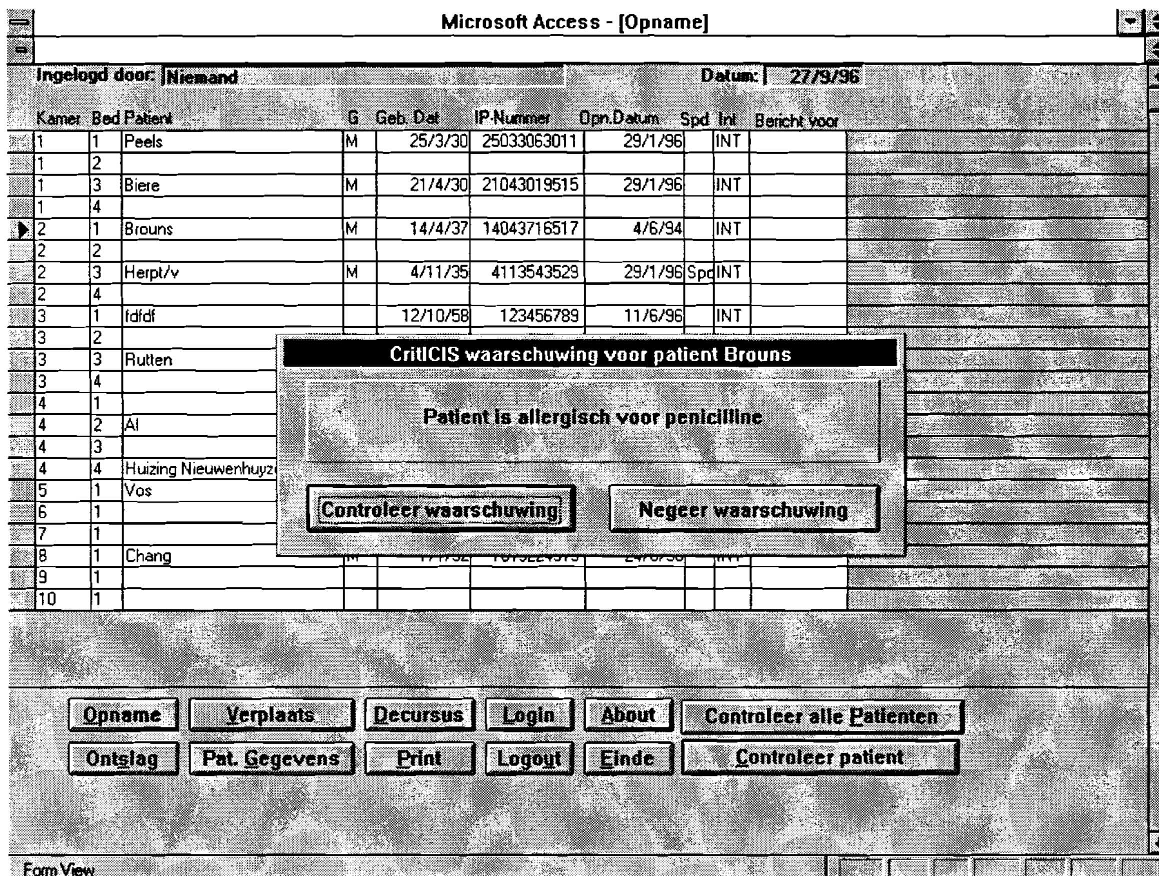

Figure 5.1: A CritICIS warning

Whenever a user of ICIS wants to check a patient (for example, after administering the patient), the user presses the 'Check patient' button ('Controleer patient' in Dutch). For example, the user that operates ICIS in figure 5.1, wants to check the patient, named Brouns, for inconsistencies. In order to accomplish this, the user selects the patient in ICIS (indicated by the arrow before the patient's name) and presses the 'Check patient' button, indicating that CritICIS must be activated to check

the selected patient. As a result, ICIS activates CritICIS by sending a 'wake-up' signal, along with the admission number of the selected patient (in this example, the admission number of the patient, named Brouns) and some additional parameters. When activated, CritICIS gathers the required data for patient Brouns, supplies it to the SIMPLEXYS inference engine and activates the inference engine (see section 5.3.2). The SIMPLEXYS inference engine then processes the knowledge base, containing the rules that describe the various possible inconsistencies (see section 5.4). When an inconsistency is found, it is provided to the user as a warning. In order to integrate the supplied warnings into the user interface of ICIS, the warnings are implemented as so-called pop-up windows, floating above ICIS. An example is the warning, shown in figure 5.1. This warning suggests to stop administering penicillin to patient Brouns, because this patient is allergic to penicillin.

After the warning is shown to the user, the user has two options, indicated by the two buttons in the pop-up window:

- Ignore the warning ('Negeer waarschuwing' in Dutch). When the user chooses this option, the pop-up window disappears and the next warning is shown (if available). However, the next time this patient is checked, the warning is shown again.
- Check the warning ('Controleer waarschuwing' in Dutch). When the user selects this option, a new pop-up window appears (figure 5.2). The difference between this pop-up window and the pop-up window in figure 5.1, is that the left button is changed into a 'This warning is correct' button ('Deze melding is terecht' in Dutch) and the right button has now become a 'This warning is incorrect' button ('Deze melding is onterecht' in Dutch). The second pop-up window always stays on top of the screen and its purpose is too able the user to mark the given warning as correct or incorrect. To accomplish this, the user checks the warning by examining the data in ICIS.

Microsoft Access - [Medicatie]

Patient: 14043716517 Brouns

Medicijn	Freq.		Eenheid + Farm. vorm		Toedienings weg	Begin		Eind	Dag	14	16	18	20	22	24	2	4	6	8	10	12
Penicilline	2	X DGS	4	µg/tabl	oraal	8/10/96	KTN		0		X								X		
Amoxicilline	4	X DGS	1000	mg	i.v.	6/6/96	KTN		124			X			X			X			X
Digoxine	1	X DGS	0.25	mg	i.v.	6/6/96	KTN		124										X		
Diltiazem	3	X DGS	120	mg/tablet	oraal	4/6/96	KTN		126		X				X				X		
Amoxicilline	4	X DGS	1000	mg	i.v.	6/6/96	KTN	6/6/96	0			X			X			X			X
Penicilline	2	X DGS	4	µg/tabl	oraal	6/6/96	KTN	8/10/96	124		X								X		
Penicilline	2	X DGS	4	µg/tabl	oraal	8/10/96	KTN	8/10/96	0		X								X		
	0	DAG	0			8/10/96			0												

CritICIS waarschuwing voor patient Brouns

Patient is allergisch voor penicilline

Deze melding is terecht | Deze melding is onterecht

Menu: Barcodes, Complicatie, (Be)Handeling, Onderzoek, Medicatie, Transf/Voed, Verslagen, Verslag zien, Kliniek, Lab, SOP, TISS

Toevoegen Medicatie | Stop Medicatie | Pompdosering | Klaar

Wijzigen Medicatie | Info

Form View

Figure 5.2: Marking a warning as right or wrong

For example, the user in the above-mentioned example has selected the medication table to check whether patient Brouns has received penicillin. The top row of the medication table in figure 5.2 shows that this is the case. After checking whether the patient is also allergic to penicillin (by examining the allergies table) the user then marks this warning as correct or incorrect by pressing the corresponding button. The pop-up window then disappears and the next warning is presented (if there is one). The acquired information about this warning is stored into a database in order to measure the performance of the warning (see section 5.5.2).

Another possibility for the users of ICIS is to check all the patients that are currently administered, by means of the 'Check all patients' button ('Controleer alle Patienten' in Dutch). When this button is pressed, CritICIS is activated again by ICIS and checks all the administered patients, also presenting the found inconsistencies as warnings.

Finally, it is also possible to activate CritICIS automatically whenever a table in ICIS has changed. However, this feature of CritICIS is disabled by default, in order to increase the execution time of ICIS (section 5.3.2).

5.2. An overview of CritICIS

Figure 5.3 shows an overview of the internal structure of the critiquing system.

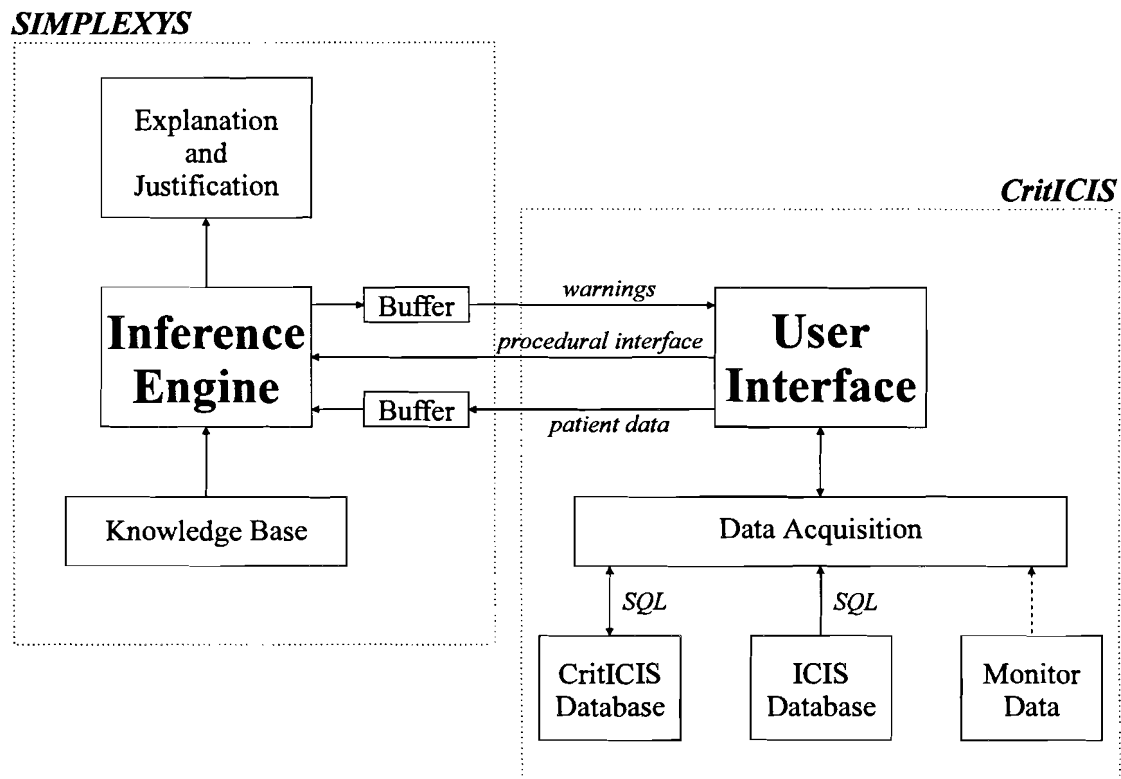

Figure 5.3: The various components of CritICIS

In the remains of this chapter, the term CritICIS is used ambiguously. It normally represents the entire critiquing system, but it may also refer exclusively to the collection of components that is responsible for the user interface and the data acquisition (figure 5.3). In order to prevent vagueness, the module, containing the user interface and data acquisition components is referred to as the *CritICIS module* or the *main application*, whereas the critiquing system itself is called the *CritICIS system* or just CritICIS.

The CritICIS module is the critiquing system's main module. It is implemented (using Delphi) as a Microsoft windows application and contains two elements: the user interface and the data acquisition components.

The second module, containing the SIMPLEXYS inference engine component (called the *SIMPLEXYS module*) is separated from the CritICIS module in order to simplify the maintenance of the knowledge base as well as the components of the main application. Communication between the two modules is realized by means of a *procedural interface* and two buffers that are accessible by both the SIMPLEXYS module and the CritICIS module.

5.2.1. The user interface

The user interface is the core of the CritICIS module and provides the user with a means of communication. Considering two types of users (*daily users* on the one hand and *system developers* and *knowledge engineers* conversely), the user interface consists of two screens:

- **The warning screen**: The warning screen is the only means of communication between the critiquing system and the daily users of ICIS (usually the medical staff). Whenever CritICIS is activated by ICIS, this screen displays the warning messages (in Dutch), reported by the SIMPLEXYS module. The warning screen is implemented as a pop-up window, floating above ICIS (figure 5.1 and figure 5.2). As a result, from the viewpoint of the daily users, the critiquing system behaves like an integrated part of ICIS.
- **The debugging screen**: This screen displays the current status of the critiquing system and is divided into two windows, both used for debugging purposes (figure 5.3). The top window displays the currently checked patients (represented by their admission numbers) and the warnings that were reported by the SIMPLEXYS module (in Dutch), whereas the bottom window displays the patient's data, extracted from the ICIS database. The information in the debugging screen is useful for both system developers and knowledge engineers.

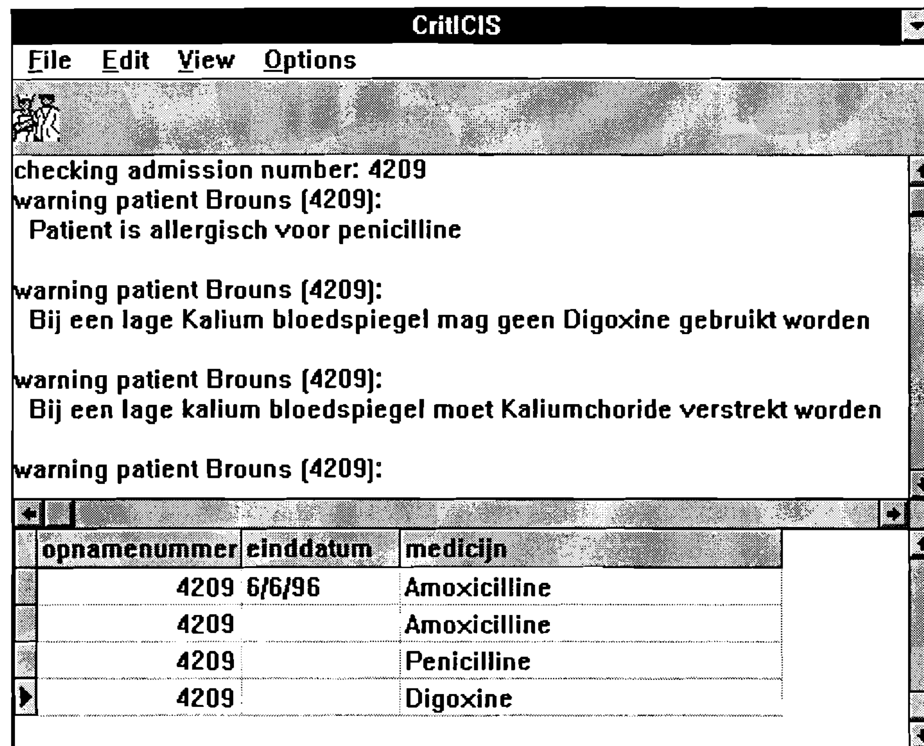

Figure 5.4: The debugging window of CritICIS

System developers are able to verify the system's functioning, whereas knowledge engineers may check the performance of the various rules in the knowledge base. When CritICIS is activated by ICIS, the debugging screen is initially hidden from the user.

5.2.2. The data acquisition components

The critiquing system (partially) has access to three sources of information: the ICIS database, the SDN network and the critiquing system's own database, called the *CritICIS database* or *ruleinfo database*. The ICIS database and the SDN network contain the necessary data, required to supply decision support to the medical staff, whereas the ruleinfo database contains information about the performance of the various rules in the knowledge base.

At present, the data acquisition element consists of two components, `ICISData` and `ruleData`. The `ICISData` component reads the data from the ICIS database and the `ruleData` component is able to read data from and write data to the CritICIS database.

Using the inheritance technique, both the `ICISData` and the `ruleData` component are derived from the `TData` base class. This class is a Delphi component, designed to make a connection with a specified database. The `TData` class is not a fully functional component. For example, it is not possible to open, close, read data from or write data to a database, using the `TData` class. The only task, the `TData` class can to perform, is establishing a connection with a specified database by means of setting various properties, such as:

- `aliasName`: this is the *alias* of the actual database. The `TData` class communicates with databases through the so-called *Borland Database Engine* (BDE), a communication layer between a Delphi application and a database. Every database, accessed through BDE is identified by an alias, which is stored into the `aliasName` property.
- `keepConnection`: a Boolean property, indicating whether the connection between the `TData` component and the actual database closes whenever there is no data transfer between the component and the database. When the `keepConnection` property is set to True, the rate of the data transfer increases but other programs may not be able to communicate with the database at the same time.
- `loginPrompt`: this Boolean property indicates whether the user must provide a login name and a password when connecting to the database.

In order to read data from and write data to a database, the `TData` class must be extended with methods that are able to perform these tasks. Examples of these extended classes are the `ICISData` class and the `ruleData` class. The `ruleData` class is the simplest of the two components. Besides the properties, inherited from the `TData` component, the `ruleData` class contains two additional methods, namely:

- `openQueries`: this method first establishes a connection with the database, indicated by the `aliasName` property, after which all the required data is gathered by means of a *Structured Query Language* (SQL) statement. SQL is a relational

- `openQueries`: this method first establishes a connection with the database, indicated by the `aliasName` property, after which all the required data is gathered by means of a *Structured Query Language* (SQL) statement. SQL is a relational database language, used to define, manipulate, search and retrieve data in databases. For example, the SQL statement `SELECT * FROM ruleinfo` fetches all the data from the CritICIS database.

- `closeQueries`: the method closes the connection between the `ruleData` component and the CritICIS database, preserving the actual link between the component and the database when the `keepConnection` property is set to True.

The ruleData component also contains a second component, called `TICISQuery`. This component holds the actual data, gathered by the `openQueries` method. Additionally, the `TICISQuery` component contains several methods to obtain the data that is stored in the component. An overview of the `TICISQuery` component, along with the other developed Delphi components, is presented in Appendix B.

The `ICISData` component is more complicated than the `ruleData` component. Similar to the `ruleData` component, the `ICISData` component also contains a `openQueries` function. However, the `openQueries` function also accepts a patient's admission number as a parameter. For example, calling the method `openQueries(4209)` opens a connection with the ICIS database and fetches all the data of the patient with admission number 4209, also by means of SQL-statements.

Another difference between the two descendants of the `TData` class, is that the `ICISData` class communicates through two communication layers with the ICIS database. Besides the above-mentioned BDE layer, `ICISData` also utilizes the so-called *Open DataBase Connectivity* (ODBC) protocol. This protocol enables an application to establish a connection with a database, regardless of its nature. For example, through ODBC, it is possible to establish a connection with a Microsoft Access database, an Oracle database or a FoxPro database, without the necessity of adapting the used SQL statements. A disadvantage of ODBC is a considerable time

delay when reading data from the database. However, ODBC is the only means of communicating with the ICIS database by means of SQL statements.

Besides the `openQueries` and `closeQueries` procedures, the `ICISData` component contains various methods to search in, read from and write to the ICIS database:

- `FindFirst`: this Boolean function searches for patient data in a specified ICIS table. For example, `FindFirst(5, 4209)` searches for the first record in the medication table that contains all the medication, administered to the patient with admission number 4209 (the first parameter in the `FindFirst` procedure is a integer that stands for a certain ICIS table. For example, 5 represents the medication table. The use of these numbers is more thoroughly explained in section 5.3.2). If the specified table contains no records, regarding that patient, the function returns False.
- `FindNext`: this is also a Boolean function, searching for the next record, indicated by the `FindFirst` function. When no additional records are found, the `FindNext` function returns False.
- `Next`: this procedure jumps to the next record in a specified table. For example, `Next(5)` jumps to the next record in the medication table.
- `GetField`: this function returns a certain field from the current record in a specified table. For example, `GetField(5, admissionDate)` returns the date, the medication in the current record was administered.

The `ICISData` contains more functions and procedures, used for communication with the various tables in the ICIS database. An overview of this component is shown in Appendix B.

The third module, called `monitorData`, is still in the development stage. As a result, obtaining data from the SDN network is not possible at the moment.

5.2.3. The SIMPLEXYS module

This module contains the SIMPLEXYS inference engine. It is implemented as a Delphi component and contains several methods and properties to communicate with the outside world (see also Appendix B). When activated by means of the procedural interface, the inference engine performs various actions, such as:

- Create two buffers, used for communication with the CritICIS module.
- Read the patient's data from ICIS (supplied by the CritICIS module).
- Execute the production rules in the knowledge base.
- Report detected warnings to the CritICIS module.
- Write debug information to various files. This information is afterwards used for explanation and justification purposes.

Usually, Delphi components are all integrated into one Delphi application. For example, the main application contains several components, such as the data acquisition components and various visual elements. However, in order to simplify the maintenance of both modules, the SIMPLEXYS inference engine component is separated from the other components. It is embedded in a *Dynamic-Link Library* (DLL) and communicates with the CritICIS module by means of the procedural interface and two buffers, both created by the SIMPLEXYS module. The first buffer contains the patient's data and is implemented as a list of records (figure 5.5).

This list, called `dataList`, consists of nine records, corresponding the nine ICIS tables that are currently implemented. Besides a pointer to the next element, each record contains three fields:

- `name`: This field contains the name of the record, equal to the name of the corresponding ICIS table.
- `changed`: This field indicates whether the record's information has changed since the last time CritICIS checked this patient.

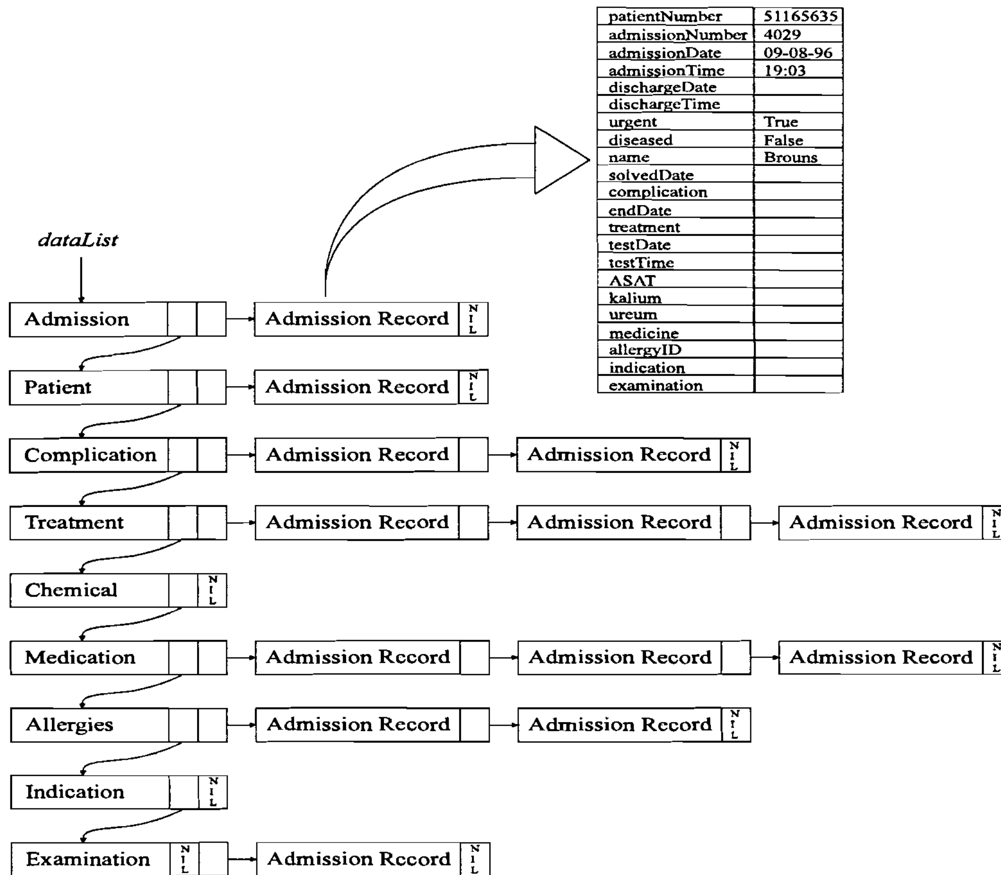

Figure 5.5: Structure of the buffer, containing patient data

- `patientData`: This field points to a list of so-called *admission records*. Each admission record contains information, extracted from the corresponding ICIS table. For example, the `patientData` field of the 'admission' element, points to a list of admission records with the patient's admission data, extracted from the admission table. This data includes information such as the patient's IP-number, admission number and admission date and time. Another example is the 'medication' record that contains a list of the patient's medications.

Since each admission record is composed of identical fields, some fields are not utilized (see for example the singled out admission record in figure 5.5, where only 7 of 22 fields are filled with data). Although this implementation uses more memory than necessary (negligible, however, compared to the total amount of required memory), it considerably simplifies the buffer's structure and the adding of new tables.

The `dataList` list is implemented as a Delphi component and communicates with the outside world by means of various methods, such as `AddElement` (add a new admission record to a list), `GetElement` (get the contents of a admission record) and `GetCount` (get the number of admission records in a list).

The other buffer, containing the warning messages generated by the SIMPLEXYS inference engine, is also implemented through lists (figure 5.6).

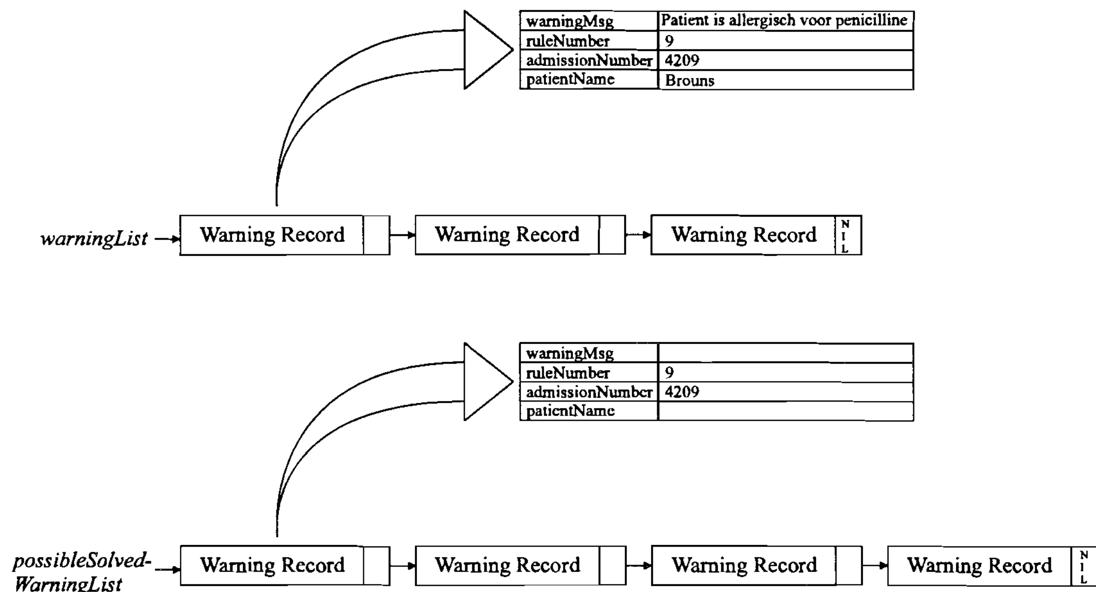

Figure 5.6: Structure of the buffer, containing warnings and possible solved warnings

This buffer consists of two lists, `warningList` and `possibleSolvedWarningList`. The `warningList` list contains warning messages that are generated by the SIMPLEXYS inference engine and shown to the user through the warning screen. The properties of every warning are stored in a so-called *warning record*, which contains four fields:

- `warningMsg`: This field contains the text of the warning message.
- `ruleNumber`: Every warning is identified by means of its *identification number* or *rule number*. This number is stored into the `ruleNumber` field.
- `admissionNumber`: This field contains the admission number of the patient that received the warning.

- `patientName`: This field contains the name of the patient with the above-mentioned admission number. The purpose of this field is to show the patient's name to the user (along with the text of the warning message).

The second list, called `possibleSolvedWarningList`, consists of a list of possible solved warnings. Whenever the SIMPLEXYS inference engine reports a warning, CritICIS stores it, along with the admission number of the currently checked patient, into the CritICIS database (see also section 5.3.2).

When this warning is not issued during the next time CritICIS is executed (probably to check another patient), the SIMPLEXYS module adds it to the possible solved warning list, after which CritICIS compares the admission numbers and the rule numbers in the possible warnings list's records with the admission numbers and the rule numbers in the CritICIS database. When equal, the warning in the CritICIS database is marked as solved. Using this procedure, CritICIS is able to detect whenever a reported warning is solved by the users of ICIS.

In order to simplify the buffer's structure, the possible solved warning list is also composed of warning records. However, only the `ruleNumber` field and the `admissionNumber` field are used, because the warning message and the patient's name are not relevant when checking for solved warnings.

Both lists are implemented as (similar) Delphi components and communicate with the outside world by means of various methods, such as `Add` (add new warning record to list), `Get` (get next warning record from list) and `Reset` (reset list). An overview of these components is shown in Appendix B.

As mentioned earlier, the SIMPLEXYS inference engine component is implemented as a DLL (Dynamic-Linked Library). As a result, besides the data transfer through the two buffers, all communication between the SIMPLEXYS module and the CritICIS module takes place by means of a procedural interface. This, because it is not possible to access a component's methods or properties in a dynamic library directly. The

SIMPLEXYS module's procedural interface consists of several pre-defined procedures (exported by the DLL), such as:

- `InferCreate`: Creates and initializes the SIMPLEXYS inference engine component.
- `InferGetDataList`: Obtains the address of the buffer, containing the patient's data by means of calling the `GetDataList` method of the SIMPLEXYS inference engine component.
- `InferGetErrorList`: Obtains the address of the buffer, containing the warnings, using the `GetErrorList` method.
- `InferGetPossibleSolvedErrorList`: Obtains the address of the buffer, containing the possible solved warnings, using the `GetPossibleSolvedErrorList` method.
- `InferStartUp`: Initializes the inference engine with the `StartUp` method.
- `InferFirstRun`: Executes the first run of the inference engine through the `FirstRun` method.
- `InferNextRun`: Executes the next run of the inference engine through the `NextRun` method.
- `InferCloseDown`: Closes the inference engine down, using the `CloseDown` method.
- `InferDisable`: Aborts the inference engine.
- `InferIsEnabled`: Indicates if the inference engine is activated by means of checking the `enabled` property.
- `InferSetContext`: Sets the context of the inference engine by means of setting the `context` property.
- `InferGetContext`: Obtains the context of the inference engine through the `context` property.
- `InferFree`: Deletes the inference engine component.

Although these procedures are often nothing more than a wrapper around the component's methods, it is the only means of communicating with the SIMPLEXYS

module. The purpose of the above-mentioned procedures is explained more thoroughly in the remains of this chapter.

5.3. Executing the critiquing system

One of the initialization tasks of ICIS is to launch the critiquing system. As a result, whenever ICIS is executed, CritICIS is also launched. As an alternative, it is also possible to launch CritICIS manually. This may be useful for system developers to test the system's performance.

5.3.1. Initializing CritICIS

When CritICIS is executed, it performs several initialization tasks, shown in listing 5.1 (presented in pseudo-code).

```
procedure Initialize;
var
  dataList, {address of list with patient data}
  ICISList: pointer; {address of list, containing messages from ICIS}
begin
  {load the SIMPLEXYS library}
  LoadLibrary(SIMPLEXYS);

  {create inference engine and obtain address of list, containing
   patient data}
  InferCreate;
  dataList := InferGetDataList;

  {create list, containing messages from ICIS}
  ICISList.Create;

  {initialize user interface and hide debugger window}
  InitializeUserInterface;
  debuggerWindow.Minimize
end;
```

Listing 5.1: Initialization tasks of CritICIS

First, the dynamic library, containing the SIMPLEXYS inference engine component is loaded, after which the inference engine is created and the address of the **dataList** component is obtained (and stored into a variable for further use). Next, the so-called **ICISList** list is created. This list contains messages, implemented as Pascal strings, from ICIS (explained in the next section). Finally, the user interface is initialized and

the debugger window is hidden from the user. After these initialization tasks, CritICIS resides into the background until activated by ICIS.

5.3.2. Activating CritICIS

All communication between ICIS and CritICIS takes place through DDE (Dynamic Data Exchange), a Microsoft Windows protocol, used for sending data to and receiving data from other applications. To activate CritICIS, ICIS passes a Pascal string to the critiquing system by means of executing a so-called *DDE-macro*, exported by CritICIS. The Pascal string is supplied as the macro's parameter, which is then added by the macro to a list, called `ICISList` (figure 5.7)

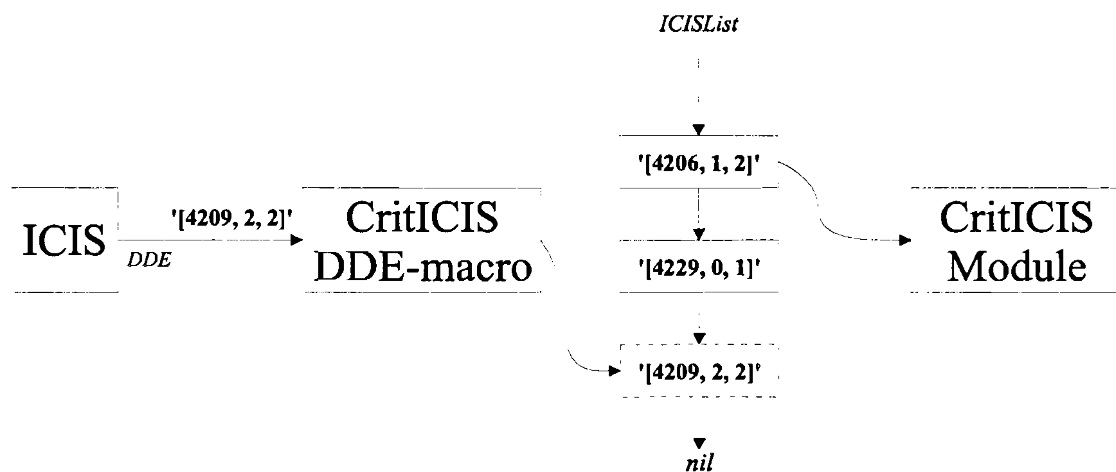

Figure 5.7: Communication between ICIS and CritICIS

The CritICIS module continuously checks this list whether it is empty. If not, the first element is extracted, after which CritICIS fetches the necessary data and executes the SIMPLEXYS inference engine, based on the obtained string.

As shown in figure 5.7, the Pascal string consists of the following syntax:

'[' + *admission number* + ',' + *table number* + ',' + *context number* + ']'

Besides the start marker ('['), the end marker (']') and the separators (','), the string is composed of three integer values:

- **The admission number**: The string's admission number is the admission number of the current selected patient.
- **The table number**: The table number indicates which ICIS table has been changed since the last time the selected patient was checked by CritICIS. Using this value, it is possible to limit the execution time of the SIMPLEXYS inference engine by means of solely executing the rules that require information from the table, represented by the table number. For example, in order to check the warning '*A high ASAT value may indicate a myocardial infarction*', CritICIS requires the patient's current ASAT value (stored into the chemical table) as well as a list of the patient's complications (stored into the complication table). Only when the table number matches that of the chemical table or the complications table, the warning is checked (through firing the rules that make up the warning). An overview of the available ICIS tables with corresponding table numbers is shown in table 5.1.

Table number	ICIS table
0	All tables
1	Admission table
2	Complication table
3	Treatment table
4	Chemical table
5	Medication table
6	Allergies table
7	Indication table
8	Patient table
9	Examination table

Table 5.1: Table numbers of the various tables in ICIS

To indicate that all tables in ICIS have changed, the table number is set to zero. As a result, CritICIS will check all warnings in the knowledge base (although the actual number of checked warnings also depends on the current context).

- **The context number**: The context number indicates in what manner CritICIS is activated by ICIS. Currently, it is possible to activate the critiquing system in three ways:
  - By means of selecting a patient and pressing the 'check patient' button in ICIS (CritICIS then operates in the *demand* context).
  - By means of an *event* from ICIS. This event takes place whenever the data in an ICIS table has changed. However, in order to react to these events, CritICIS must operate in the so-called *online* mode. When CritICIS is running into the *offline* mode, these events are ignored. By default, CritICIS starts in the offline mode, in order to save time (the processing of these events usually takes some time, due to the slow execution of the SQL statements). When ICIS has been activated through an event from ICIS, it operates in the *event* context.
  - By means of a *timer event* or by pressing the 'check all patients' button. When CritICIS is activated in this manner (called the *interval* context), it first adds all the administered patients to the `ICISList` list, after which every patient is checked sequentially. The interval context was initially meant for checking all patients in ICIS in regular time intervals (for example, once a day), but it is also possible to activate CritICIS in this context manually, by means of the 'check all patients' button in ICIS.

Table 5.2 shows an overview of the above-mentioned contexts, along with the corresponding context numbers.

Context number	Context
1	Demand context
2	Event Context
3	Interval context
4	End CritICIS

Table 5.2: Contexts in CritICIS

The last entry in table 5.2 (context number 4) is used to end the CritICIS application. Whenever ICIS shuts down, it calls the DDE-macro with context number 4 in order to shut down CritICIS as well.

It is possible to define a set of warnings for each context separately. For example, whenever CritICIS is activated in the demand context or in the interval context, all warnings are checked (depending on the table number). However, when CritICIS operates in the event context, only the warnings, regarding the patient's allergies are checked, in order to limit the execution time of the inference engine. More information about the assigning of warnings to the various contexts is found in section 5.3.2.

Figure 5.6 shows some examples of Pascal strings that are passed from ICIS to CritICIS. For example, the first string in the list ('[4206, 1, 2]') instructs CritICIS to check the patient with admission number 4206 in the event context (context number 2). Also, only the warnings that require information from the admission table (table number 1) are to be checked. The second element in the list ('[4229, 0, 1]') indicates that the user of ICIS selected the patient with admission number 4229 and pressed the 'check patient' button (it is common that the table number is set to 0 when CritICIS is activated in the demand context, ensuring the checking of all warnings). Whenever CritICIS is activated in the interval context, the admission number is ignored, because the critiquing system itself adds all the administered patients to the `ICISList` list.

When available, CritICIS reads the first element of the `ICISList` list and extracts the admission number, table number and context number from the element. Based on these values, CritICIS fetches the required patient data and executes the SIMPLEXYS inference engine (listing 5.2).

```
procedure Activate
const
  Event = 2; {context number of event context}
var
  ICISListElement: string; {first element from list}
  admissionNumber,
  tableNumber,
```

```
  contextNumber: integer; {values, extracted from ICISListElement}
begin
  {look if list is empty}
  if ICISList.count <> 0 then {list is not empty}
  begin
    {Get first element and extract admission number, table number and
    context number}
    ICISListElement:= ICISList.Items[0];
    ICISList.Delete(0);
    ExtractValues(ICISListElement, admissionNumber, tableNumber,
                  contextNumber);

    {abort if CritICIS is offline and context is 'event'}
    if (offline = False) or (context <> Event} then {continue}
    begin
      {Read patient data and supply it to the SIMPLEXYS module}
      FillDataList(admissionNumber, tableNumber)

      {activate SIMPLEXYS inference engine, set context and initiate
       first run}
      InferStartUp;
      InferSetContext(contextNumber);
      if InferFirstRun = Ready then {SIMPLEXYS only needed one run}
        {close down SIMPLEXYS inference engine}
        InferCloseDown
    end
  end
end;
```

Listing 5.2: Starting up the inference engine

In order to store the patient data into the **dataList** structure (figure 5.5), the **ICISData** component is utilized (section 5.2.2). By means of the component's various methods, patient data is extracted from all tables and stored into the **dataList** list (listing 5.3).

```
procedure FillDataList(admissionNumber, tableNumber: integer)
const
  highestTableNumber = 9; {highest table number, see table 5.1}
var
  i: integer {counter}
  nextAdmissionRecord: pointer; {pointer to admission record}
begin
  {establish link with ICIS Database}
  ICISdata.OpenQueries(admissionNumber);

  {process all currently implemented tables in ICIS}
  for i:=1 to highestTableNumber do {process next ICIS table}
  begin
    {read first record with patient data from current table, if found}
    if ICISData.FindFirst(i, admissionNumber) = True then
    {first record found}
    begin
```

```
    {create new admission record, fill it with patient data from all
     required fields and append record to dataList}
    nextAdmissionRecord := CreateNewAdmissionRecord;
    FillAdmissionRecord(i, nextAdmissionRecord);
    dataList.AppendAdmissionRecord(i, nextAdmissionRecord);

    {process other records in table, if any}
    while ICISdata.FindNext = True do {there is still a next record}
    begin
      NextAdmissionRecord := CreateNewAdmissionRecord;
      FillAdmissionRecord(i, nextAdmissionRecord);
      dataList.AppendAdmissionRecord(i, nextAdmissionRecord)
    end
  end
 end;

 {dataList list filled, set table number of changed table}
 dataList.SetTableNumber(tableNumber);

 {close connection with ICIS database}
 ICISdata.CloseQueries
end;
```

Listing 5.3: Collecting and storing the patient's data

For each table in ICIS, all records that contain data, concerning the patient, specified by the **admissionNumber** parameter, are fetched through the **ICISData** component. For each record, information from the required fields is extracted through the **FillAdmissionRecord** procedure and stored into the **dataList** list. In order to acquire the data from the required fields, the **FillAdmissionRecord** procedure internally calls the **GetField** procedure, described in section 5.2.2. However, the number of required fields depends on the current ICIS table. For example, the required fields of the admission table are the **patientNumber** field, the **admissionNumber** field, the **admissionDate** and **admissionTime** field, the **urgent** field, the **diseased** field and the **patientName** field (see also figure 5.5), whereas the required fields of the medication table are the **admissionNumber** field, the **medicationName** field and the **endDate** field (containing the date, medication treatment was ended). For this reason, the number of the current table is supplied to the **FillAdmissionRecord** procedure, which is internally translated into the amount of required fields.

After creating and filling the admission record, it is appended to the corresponding element of the **dataList** list by means of the **AppendAdmissionRecord** method. In

order to append the admission record to the right element, the number of the current table is also supplied as a parameter (which is internally translated to the corresponding element). Finally, the table number of the changed table is set and the connection with the ICIS database is terminated.

Whenever the first run of the SIMPLEXYS inference engine was also the last one, the inference engine is deactivated by the **Activate** procedure. However, a typical SIMPLEXYS expert system (including CritICIS) usually requires more than one run. In order to execute further runs, CritICIS utilizes the so-called **Idle** procedure. This procedure is (continuously) executed by the operating system whenever the processor is not used by another program. An overview of the **Idle** procedure (including **Activate**) is shown in listing 5.4.

```
procedure Idle
begin
  {look if inference engine is already activated}
  if InferIsEnabled = False then {inference engine is not activated}
    Activate

  {look if it is necessary to execute a next run}
  if InferIsEnabled = True then {inference engine is still activated}
    if InferNextRun = Ready then {this was the last run}
      InferCloseDown
end;
```

Listing 5.4: The Idle procedure of CritICIS

The first action of the **Idle** procedure is to check whether the inference engine is still executing runs. If not, the **Activate** procedure is called to extract the next element from the **ICISList** list (if possible). Next, the **Idle** procedure executes the inference engine's next run, if necessary. If this run was also the last one, SIMPLEXYS is closed down, automatically setting the **enabled** property to False.

By means of the **Idle** procedure, it is possible to execute CritICIS in multitasking environments (for example in the background). However, in such an environment, the **Idle** procedure must be considered a *critical section*. Otherwise, sequential runs may overlap each other. At the moment, this is not a pressing problem, because the operating system, currently used (Microsoft Windows 3.1x), utilizes the so-called

cooperative multitasking algorithm, making it impossible for a procedure to overlap itself.

5.3.3. Processing the warnings

After the inference engine is closed down, the CritICIS module processes the **warningList** list, containing warnings, reported by the inference engine (listing 5.5).

```
procedure ProcessWarningList
const
  {conclusions}
  Correct = 1;
  Incorrect = 2;
  Ignored = 3;
var
  warningList: pointer; {pointer to list with warning messages}
  i, {counter}
  admissionNumber, {admission number of selected patient}
  ruleNumber, {number of issued warning}
  conclusion: integer; {conclusion of user that checked warning}
  warningMsg, {text of warning message}
  patientName: string; {name of selected patient}
begin
  {get address of warning list}
  warningList := InferGetErrorList;

  {process all warnings in list}
  for i:=0 to warningList.count-1 do
  begin
    {extract fields from warning record}
    ExtractFields(warningList.Items[i], warningMsg, ruleNumber,
                  admissionNumber, patientName);

    {show warning to user: return code determines conclusion (Correct,
     Incorrect or Ignored)}
    conclusion := ShowWarningScreen(warningMsg, patientName);

    {store warning, along with conclusion into CritICIS database}
    ruleData.Add(admissionNumber, ruleNumber, conclusion)
  end
end;
```

Listing 5.5: Processing the warning messages

After obtaining the address of the warning list, the **ProcessWarningList** procedure processes each element sequentially. First, the four fields of the currently processed record are extracted, after which the procedure shows the warning screen to the user (figure 3.2). Using ICIS, the user then evaluates the warning, marking it Correct (the

warning was correct), Incorrect (it was a false alarm) or Ignored (the warning was ignored by the user and will be shown again, the next time CritICIS checks this patient). Finally, this evaluation or *conclusion* is stored into the CritICIS database, along with the patient's admission number and the number of the warning, using the **ruleData** component (described in section 5.3.2). A section of the CritICIS database is shown in table 5.3.

Warning number	Admission number	Rule number	Conclusion
1	4129	1	3
2	4129	7	2
3	4200	1	1
4	4200	4	4
5	4204	1	3
6	4204	1	4
7	4206	4	1
8	4206	8	1
9	4208	2	1
10	4209	1	4

Table 5.3: A section of the ruleinfo database

Besides processing the warning messages, CritICIS also checks for warnings that may be solved, using the **possibleSolvedWarningList** (listing 5.6).

```
procedure ProcessPossibleSolvedWarningList
const
  {conclusions}
  Correct = 1;
  Incorrect = 2;
  Ignored = 3;
  Solved = 4;
var
  possibleSolvedwarningList: pointer; {pointer to list with possible
                                        solved warnings}
  i, {counter}
  admissionNumber, {admission number of selected patient}
  ruleNumber, {number of issued warning}
  conclusion: integer; {conclusion of previously issued warning}
begin
  {get address of warning list, containing possible solved warnings}
```

```
 possibleSolvedWarningList := InferGetPossibleSolvedErrorList;

 {process all warnings records in list}
 for i:=0 to possibleSolvedWarningList.count-1 do
 begin
   {extract fields from warning record}
   ExtractFields(PossibleSolvedWarningList.Items[i], ruleNumber,
                 admissionNumber);

   {search for warnings in database with equal rule numbers and
    admission numbers}
   if ruleData.Find(admissionNumber, ruleNumber) = True then
   begin {get conclusion of previously issued warning}
     conclusion := ruleData.GetConclusion(admissionNumber,
                                          ruleNumber);

     {mark entry as solved, if conclusion was Correct or Incorrect}
     if (conclusion = Correct) or (conclusion = Incorrect) then
     begin
       conclusion := Solved;
       ruleData.Edit(admissionNumber, ruleNumber, conclusion)
     end
   end
 end
end;
```

Listing 5.6: Checking whether warnings have been solved

Similar to the **ProcessWarningList** procedure, the **ProcessPossibleSolved-WarningList** procedure first requests the address of the required list, after which every warning record in the list is processed sequentially. Although each warning record contains four fields, only two are utilized, namely the fields, containing the admission number and the rule number. After these fields are extracted from the warning record, the **ProcessPossibleSolvedWarningList** procedure first looks in the CritICIS database for an entry with matching admission number and rule number. If the procedure finds such an entry, indicating that this warning was issued before, it also requests the corresponding conclusion. If the conclusion was previously set to Correct or Incorrect, the conclusion is marked as Solved.

After processing the **warningList** and the **possibleSolvedWarningList**, both lists are cleared by the CritICIS module, along with the list, containing the patient data (**dataList** list), after which CritICIS extracts the next element from the **ICISList** list, if available.

5.3.4. Ending CritICIS

CritICIS ends whenever ICIS executes the DDE-macro while setting the context number to 4 or whether the user presses CTRL-X when CritICIS is active (only meant for system developers). Before CritICIS ends, it first performs some exit code (listing 5.7).

```
procedure Exit;
begin
  {shut the SIMPLEXYS inference engine down, if still active}
  if InferIsEnabled = True then {SIMPLEXYS still enabled}
    InferCloseDown;

  {delete the list, containing the Pascal strings, added by the DDE-
   macro}
  ICISList.Free;

  {delete SIMPLEXYS inference engine object and unload library}
  InferFree;
  FreeLibrary(SIMPLEXYS);
end;
```

Listing 5.7: Exit code of CritICIS

First, the SIMPLEXYS inference engine is closed down, if still active. Next, the list, containing the Pascal strings from ICIS is deleted. Finally, the inference engine component is freed (automatically deleting the **dataList**, **warningList** and **possibleSolvedWarningList** lists in the two buffers), after which the SIMPLEXYS dynamic library is unloaded.

5.4. Structure of the knowledge base

The knowledge base of CritICIS currently contains 9 different warnings, encoded in 65 rules. Table 5.4 shows an overview of the current rules, along with their rule numbers. An overview of the entire knowledge base is shown in Appendix A (in Dutch).

Rule number	Description of rule
1	An abnormal admission time usually indicates an emergency
2	There is no explanation for a long administered patient
3	Patient is discharged, but still has an registered complication or treatment
4	A high ASAT value may indicate a myocardial infarction
5	Dioxin is restricted for patients with a low Kalium value
6	A low Kalium value may require administering KaliumChloride
7	A low ureum value requires a gentamycin examination
8	The patient is not allowed to receive amoxicillin, due to allergies
9	The patient is not allowed to receive penicillin, due to allergies

Table 5.4: Currently implemented rules and corresponding rule numbers

5.4.1. Warning classification

In order to structure the knowledge base, the various warnings are classified into several categories (figure 5.8).

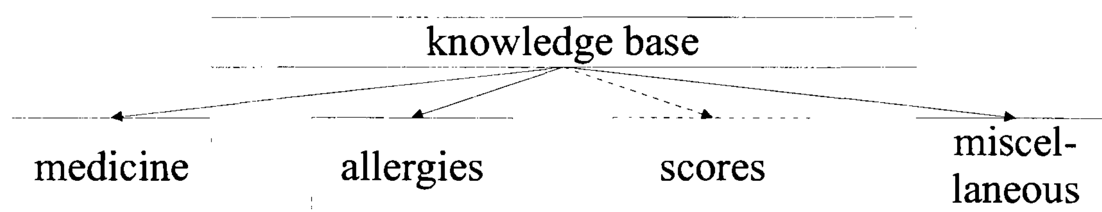

Figure 5.8: Structure of the knowledge base

As shown in figure 5.8, the warnings (also called *primary goals*) are categorized in 4 classes:

- **medicine**: warnings in this class are issued whenever the patient's condition requires a certain medicine treatment, such as administering or restricting a certain medicine. An example is the following rule, which suggests administering KaliumChloride when the patient's Kalium value is too low:

```
LOW_KALIUM_REQUIRES_KCL: 'A low Kalium value may require
administering KaliumChloride'
LOW_KALIUM and not(KCL)
```

Rule 5.1: A rule from the medicine class

- **allergies**: the warnings in the allergies category are issued when the user of ICIS administers a medicine to a patient that is allergic to that medicine. An example is:

```
PATIENT_IS_NOT_ALLOWED_TO_RECIEVE_PENICILLIN: 'The patient is not
allowed to receive penicillin, due to allergies'
ALLERGIC_TO_PENICILLIN and PENICILLIN
```

Rule 5.2: An allergy rule

- **scores**: warnings in this class are concerned with various aspects of the patient's scores, such as the APACHE score and the TISS score (see also section 3.1.1). Currently, the rules in this class are not encoded into the knowledge base, although they exist on paper.

- **miscellaneous**: this class contains miscellaneous rules, implemented at the initial stage of the knowledge base development, such as the following rule:

```
ABNORMAL_ADMINISTRATION_IS_URGENT: 'A patient that is administered
outside normal working hours, is usually marked as an emergency'
(NIGHTLY_ADMISSION or WEEKEND_ADMISSION) and not(URGENT)
```

Rule 5.3: A miscellaneous rule

Whenever a patient is administered outside normal working hours or in the weekend, it is usually marked as an emergency. The warning, encoded in rule 5.3, is issued whenever such an administration is not marked as an emergency.

In the knowledge base of CritICIS, every class is implemented as a SIMPLEXYS state rule. For example, the allergy state rule, representing the allergies class, currently contains two warnings, implemented as goals:

```
ALLERGIES_CLASS: 'Check allergies'
STATE
THEN GOAL: PATIENT_IS_NOT_ALLOWED_TO_RECIEVE_PENICILLIN,
           PATIENT_IS_NOT_ALLOWED_TO_RECIEVE_AMOXICILLIN
```

Rule 5.4: The allergies class, implemented as a SIMPLEXYS state rule

Similar, the medicine class and the miscellaneous class are also implemented, using state rules (see Appendix A).

5.4.2. Utilizing the context number

As stated in section 5.4.2, it is possible to assign a different set of warnings to each context separately, by using a combination of state rules and trigger rules. When the inference engine is executed, the system always starts in the so-called *begin context*: a state rule that is initially set to TR. Depending on the context number (which was set through the `InferSetContext` procedure), the inference engine then activates the specified context by selecting the corresponding trigger rule:

```
ON CONTEXT_NUMBER_1 FROM BEGIN TO DEMAND_CONTEXT
ON CONTEXT_NUMBER_2 FROM BEGIN TO EVENT_CONTEXT
ON CONTEXT_NUMBER_3 FROM BEGIN TO INTERVAL_CONTEXT
```

Rule 5.5: Switching to the specified context, through trigger rules

Similar, the warnings, specified for each context, are also processed by means of activating the corresponding states:

```
ON DEMAND_CONTEXT FROM DEMAND_CONTEXT TO MISC_CLASS
                                         MEDICINE_CLASS
                                         ALLERGIES_CLASS
```

Rule 5.6: Processing warnings in the demand context

As shown in rule 5.6, when CritICIS is activated in the demand context, it sequentially processes all available classes of warnings, in contrast to the event context, which only processes the warnings in the allergies class:

```
ON EVENT_CONTEXT FROM EVENT_CONTEXT TO ALLERGIES_CLASS
```

Rule 5.7: Processing warnings in the event context

Finally, when CritICIS operates in the interval context, it processes all available classes, but only if the patient is still administered:

```
ON PATIENT_ADMINISTERED FROM INTERVAL_CONTEXT TO ALLERGIES_CLASS
                                                 MEDICINE_CLASS
                                                 ALLERGIES_CLASS
ON PATIENT_DISCHARGED FROM INTERVAL_CONTEXT TO *
```

Rule 5.8: Processing warnings in the event context

As shown in rule 5.8, whenever the patient is already discharged, the inference engine deactivates all states, closing the inference process.

5.4.3. Executing warnings through strategic rules

Besides context numbers, the number of executed warnings also depends on the number of changed ICIS tables, indicated by the table number. This feature is implemented through adding a strategic rule that operates on a higher level (see also section 2.3.2):

```
FIRE_LOW_KALIUM_REQUIRES_KCL: 'Fire LOW_KALIUM_REQUIRES_KCL
warning if at least one of the required tables have changed'
(CHEMICAL_TABLE_CHANGED or MEDICINE_TABLE_CHANGED) and
LOW_KALIUM_REQUIRES_KCL
```

Rule 5.9: Adding strategic rules to the knowledge base

As stated in section 4.3.1, the **and** and **or** operators in SIMPLEXYS are conditional. As a result, the **LOW_KALIUM_REQUIRES_KCL** rule (rule 5.1) is fired only when the chemical table or the medicine table is changed. By means of adding a strategic rule to every warning, it is possible to increase the execution speed of the inference engine considerably.

5.4.4. Adding warning records by means of THELSEs

In order to add warning records to the **warningList** as well as the **possibleSolved-WarningList** list, the rule body of each warning is followed by two THELSEs:

```
LOW_KALIUM_REQUIRES_KCL: 'A low Kalium value may require
administering KaliumChloride'
LOW_KALIUM and not(KCL)
THEN DO AddToWarningList('A low Kalium value may require
                          administering KaliumChloride', 6)
ELSE DO AddToPossibleSolvedWarningList(6)
```

Rule 5.10: Expanding a warning with THELSEs

Whenever the warning evaluates as TR, the `THEN DO` part is carried out, executing a Pascal procedure with the warning message and rule number as parameters. This procedure then adds a warning record to the `warningList` list, consisting of the received parameters, along with the patient's name and admission number (obtained by the procedure itself). Similar, if the rule evaluates as FA (indicating that there was no warning), the `ELSE DO` part is carried out, resulting in the appending of a warning record to the `possibleSolvedWarningList` list (in which the fields, containing the warning message and the patient's name are omitted).

5.5. Developing CritICIS by means of rapid prototyping

Although not entirely separable, the development of CritICIS can be divided into two processes:

- Development of the CritICIS application (consisting of the CritICIS module as well as the SIMPLEXYS module).
- Building and maintaining the knowledge base.

5.5.1. Developing the CritICIS module

The CritICIS module is developed, using the tools and techniques, described in sections 3.1 (rapid prototyping) and 3.2 (Borland Delphi).

First, through interviews with the medical staff and some informal specifications on paper, a project plan was drafted (consisting of an overview of required components

and their relations, and a suitable database structure). Next, based on the project plan, an initial prototype was build and demonstrated to the users of the ICU. By means of their feedback, the prototype was evolved in several iteration steps, varied from minor enhancements ('bug fixes') to the adjustment of the project plan, which resulted in (partially) rewriting the module(s). Currently, the development of the CritICIS module is still in the prototype iteration phase.

5.5.2. Developing the knowledge base

The development of the knowledge base of CritICIS is merely begun. By means of interviews with domain experts (the medical staff and physicians of the ICU), knowledge has been elicited and encoded into the knowledge base. Similar to the development of the CritICIS module, this process also consisted of several iterations. As a result, a warning that started as a very simple rule evolved into a more complex set of rules with each iteration step, based on the feedback of domain experts (see also figure 4.2).

In order to assist the medical staff in the iteration process, a graphical overview, showing the performance of the various warnings, was implemented in CritICIS (figure 5.9).

In the graphical overview, each warning is represented by a histogram, showing the total number of times each warning was issued. The histogram also shows the number of incorrect warnings (bottom section of the histogram), correct warnings and ignored warnings (middle sections), and solved warnings (top section). For example, the warning with rule number 1 ('*An abnormal admission time usually indicates an emergency*') has been issued 7 times, including 2 false alarms, 3 correctly issued warnings, 1 ignored warning and 1 solved warning.

The graphical overview is used by the domain experts to evaluate and improve the performance of the various rules in the knowledge base. For example, if a warning's histogram consists mainly of false alarms, the corresponding rule is removed or adjusted. Using this aid, it might be possible for domain experts to implement,

evaluate and adjust their own rules. An advantage of this strategy is that a domain expert also becomes the knowledge engineer. However, this strategy must be pursued with great care, in order to prevent an inconsistent knowledge base.

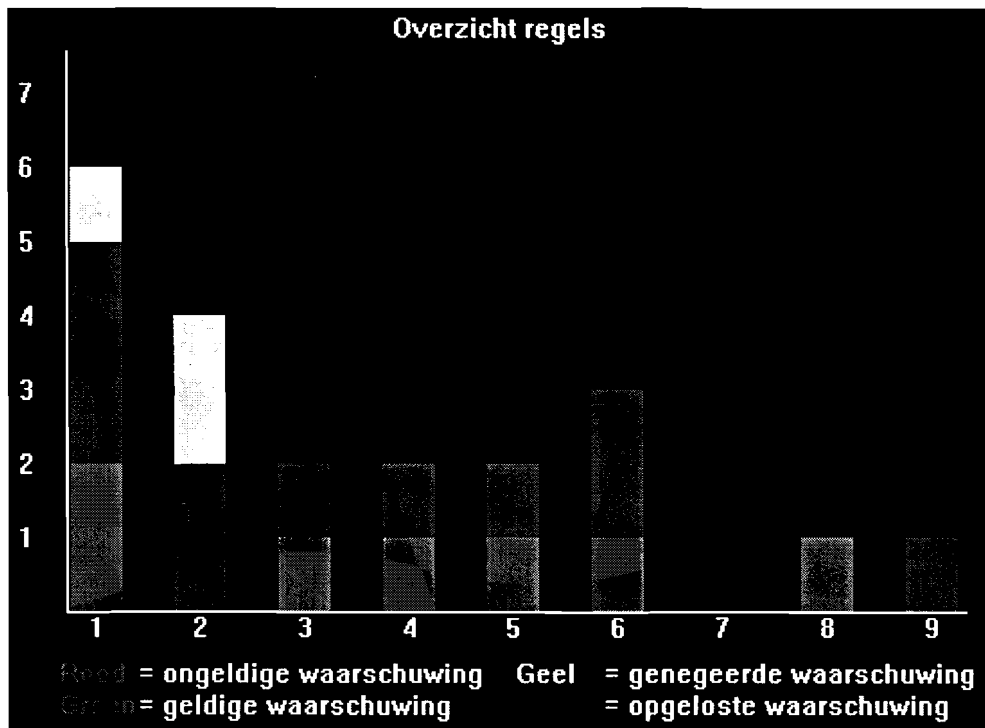

Figure 5.9: A graphical overview of various warnings

6. Conclusions

At present, CritICIS is being tested by the users of ICIS. Considering their feedback, the first results are promising: as a result of the integrated user interface of CritICIS as well as the utilized critiquing approach, the users of ICIS are satisfied with the critiquing system. Therefore, the conclusion can be drawn that it is possible to successively implement a critiquing system that provides decision support to the users of ICIS. However, in order to elaborate the functioning of CritICIS, it is very important to develop the knowledge in knowledge base. Currently, the knowledge base of CritICIS contains of rather simple validations and more interviews with the treating physicians are necessary to acquire a more 'intelligent' system.

When developing a more elaborate system, a main advantage of the already utilized critiquing approach might be the possibility of utilizing so-called *medical protocols*. Nowadays, physicians increasingly try to structure their operating procedures, writing them on paper as protocols. These protocols are well suited for implementation into the knowledge base of CritICIS, because of their formal syntax. As a result, domain experts perform their own knowledge elicitation process, probably simplifying the entire knowledge acquisition process. Domain experts from the ICU have suggested an even more straightforward knowledge acquisition process by encoding the medical protocols in the CritICIS knowledge base themselves. Although this may be a very efficient knowledge acquisition process, it must be carried out with great care, because adding new rules may produce unexpected side-effects (see also section 2.3.2).

A solution might be to develop a second version of CritICIS. The domain experts are able to add, modify and evaluate rules in this 'temporarily' version of CritICIS, after which the rules are transferred to the 'real' CritICIS system if they prove to be correct. The graphical overview, shown in figure 5.9 might be a helpful tool in order to evaluate the various rules, as well as additional programs such as the semantics checker, the protocol checker and the Tracer/Debugger. However, the current

explanation facilities of these programs are rather poor: the representation format of the data is not suitable for the domain expert and must be elaborated for this purpose.

Further, by means of the SIMPLEXYS inference engine, CritICIS operates in real-time. Common expert systems employ an inference process that performs a great deal of searching and matching, whereas the rule base of CritICIS is pre-compiled in order to prevent searching at run-time. However, this inference strategy has a significant disadvantage. The SIMPLEXYS programming language consists only of so-called propositions. As a result, it is not possible to construct 'Prolog-like' queries, such as

parent (Ad, Paul)
parent (Thea, Paul)
? **parent** (X, Paul),

which provides a list of all known parents of Paul. These queries may be useful for the search strategies, used by CritICIS (for example, to match an allergy with its restricted medicines). However, it is not possible to implement this feature into SIMPLEXYS, without holding on to the 'no-search' philosophy. A compromise might be reached by means of implementing *parameters* into the syntax of SIMPLEXYS (for example the rule `ALLERGIC_TO_PENICILLIN('Brouns')`, which means that patient Brouns is allergic to penicillin). Internally, it is possible to encode these rules into so-called hash-tables, in order to maintain the high execution speed of SIMPLEXYS. As an alternative, it might be possible to use other knowledge representations, such as frames or semantic networks.

At present, the SIMPLEXYS inference engine class must be developed again for every SIMPLEXYS application. A better solution is the design a so-called *base class*, consisting only of the methods and properties, described in 4.3.4. Whenever a new SIMPLEXYS application must be developed, the programmer uses this base class, along with the inheritance technique to create a more specific inference engine. However, this is not possible as long as the various '.qqq' files are compiled into the inference engine. A solution would be to retrieve the necessary variables and

functions at run-time, instead of compiling them into the inference engine (for example by compiling the variables and functions into a dynamic linked library). However, this requires a considerable adaptation of the SIMPLEXYS rule compiler.

Finally, although the SIMPLEXYS toolbox was not designed to develop critiquing systems, it turned out that the programming language is well suited for this approach. Especially the concept of state rules and concepts is a very powerful tool to develop well-structured critiquing systems. Also the separation of the inference engine from the user interface seems to enhance the usability of the system, as well as the real-time aspect of SIMPLEXYS, which is of great importance in dynamic environments, such as the ICU. Although there is still much research required on the development of CritICIS, especially the expansion of the knowledge base and the development of the SIMPLEXYS toolbox, CritICIS may become a very useful tool to provide decision-support in the ICU.

REFERENCES

Andriole, Stephen J. 1992. *Rapid application prototyping: the storyboard approach to user requirements analysis*. QED Technical Publishing Group.

Barr, A and Feigenbaum, E.A. 1981. *The Handbook of Artificial Intelligence. Vol. 1.* Los Altos CA: Morgan Kaufmann

Blom, JA. 1990. *The SIMPLEXYS Experiment. Real time expert systems in patient monitoring*. Eindhoven University of Technology

Chytil, MK and Engelbrecht, R. 1987. *Medical Expert Systems using personal computers*. Sigma press.

Connell, John L. and Shafer, Linda. 1989. *Structured Rapid Prototyping. An Evolutionary Approach to Software Development*. Prentice Hall.

Duda, Richard O and Shortliffe, Edward H. *Expert Systems Research*. Science 1983; Volume 220; 261-268

Groen, Marjan AH. 1995. *Technology, Work and Organization: A study of the nursing process in intensive care units*. Universitaire Pers Maastricht.

Hayes-Roth, Frederick, Waterman, Donald A and Lenat, Douglas B. 1983. *Building Expert Systems*. Addison-Wesley

Jackson, P. 1990. *Introduction to Expert Systems*. Addison-Wesley

Laffey, Thomas J, Cox, Preston A, Schmidt, James L, Kao, Simon M and Read, Jackson Y. *Real-Time Knowledge-Based Systems*. AI Magazine 1988; Volume 9, No 1; 27-45

Langlotz, Curtis P and Shortliffe, Edward H. *Adapting a consultation system to critique user plans*. International Journal of Man-Machine Studies 1983; Volume 19; 479-496

Lei, Johan van der. 1991. *Critiquing based on computer-stored medical records*. Erasmus University Rotterdam

Metnitz, PGH and Lenz, K. *Patient data management systems in intensive care - the situation in Europe*. Intensive care medicine 1995. Volume 21; 703-715.

Miller, PL. 1986. *Expert Critiquing Systems, Practice-Based Medical Consultation by Computer*. New York: Springer-Verlag.

Miller, PL.1988. *Selected topics in Medical Artificial Intelligence*. New York: Springer-Verlag.

Wyatt, Jeremy Crispin. *Clinical Data Systems, part 3: development and evaluation.* Lancet 1994. Volume 334; 1682-1688.

APPENDIX A: The knowledge base of CritICIS

RULES {rules voor de ICIS-database}

{*********************** *CATEGORIE: OVERIGE REGELS* ***************************}
ABNORMALE_OPNAMETIJD_IS_SPOED: 'Controleer CHK_ABNORMALE_OPNAMETIJD_IS_SPOED als de betreffende tabellen veranderd zijn'
OPNAME_VERANDERD and **CHK_ABNORMALE_OPNAMETIJD_IS_SPOED**

CHK_ABNORMALE_OPNAMETIJD_IS_SPOED: 'Abnormaal opnametijdstip is meestal spoed'
(not(**NORMAAL_OPNAMETIJD**) or **WEEKEND_OPNAME**) and not(**SPOED_OPNAME**)
THEN DO AddErrorMessage('Opname op abnormaal tijdstip is meestal spoed', AB_TI_SP)
ELSE DO AddSolvedErrorCheck(AB_TI_SP)

GEEN_VERKLARING_VOOR_LANGE_OPNAMEDUUR: 'Controleer CHK_GEEN_VERKLARING_VOOR_LANGE_OPNAMEDUUR als de betreffende tabellen veranderd zijn'
(**OPNAME_VERANDERD** or **BEH_VERANDERD** or **COM_VERANDERD**) and
CHK_GEEN_VERKLARING_VOOR_LANGE_OPNAMEDUUR

CHK_GEEN_VERKLARING_VOOR_LANGE_OPNAMEDUUR: 'Lange opnameduur heeft geen verklaring'
not(**PATIENT_ONTSLAGEN**) and not(**PATIENT_OVERLEDEN**) and **OPNAME_LANGER_DAN_3** and not(**COMPLICATIE** or **BEHANDELING**)
THEN DO AddErrorMessage('Een lange opnameduur moet een verklaring hebben', GE_VE_LA_OP)
ELSE DO AddSolvedErrorCheck(GE_VE_LA_OP)

PATIENT_ONTSLAGEN_MAAR_HEEFT_COMPLICATIE_OF_BEHANDELING: 'Controleer CHK_PATIENT_ONTSLAGEN_MAAR_HEEFT_COMPLICATIE_OF_BEHANDELING als de betreffende tabellen veranderd zijn'
(**OPNAME_VERANDERD** or **BEH_VERANDERD** or **COM_VERANDERD**) and
CHK_PATIENT_ONTSLAGEN_MAAR_HEEFT_COMPLICATIE_OF_BEHANDELING

CHK_PATIENT_ONTSLAGEN_MAAR_HEEFT_COMPLICATIE_OF_BEHANDELING: 'Patient is ontslagen, maar heeft nog complicatie of behandeling'
(**PATIENT_ONTSLAGEN** or **PATIENT_OVERLEDEN**) and (**BEHANDELING** or **COMPLICATIE**)
THEN DO AddErrorMessage('Patient is ontslagen of overleden, maar heeft nog steeds een compicatie of behandeling',ON_EN_CO_OF_BE)
ELSE DO AddSolvedErrorCheck(ON_EN_CO_OF_BE)

HOGE_ASAT_IS_INFARCT: 'Controleer CHK_HOGE_ASAT_IS_INFARCT als de betreffende tabellen veranderd zijn'
(**CHEMIE_VERANDERD** or **IND_VERANDERD**) and **CHK_HOGE_ASAT_IS_INFARCT**

CHK_HOGE_ASAT_IS_INFARCT: 'ASAT van meer dan 100 is waarschijnlijk infarct'
HOGE_ASAT and not(**INFARCT**)
THEN DO AddErrorMessage('Een ASAT van meer dan 100 is waarschijnlijk een infarct', HO_AS_IS_IN)
ELSE DO AddSolvedErrorCheck(HO_AS_IS_IN)

{***************** *CATEGORIE: MEDICATIE ICM TOESTAND PATIENT* *****************}
LAGE_KALIUM_SPIEGEL_VEREIST_KCL: 'Controleer CHK_LAGE_KALIUM_SPIEGEL_VEREIST_KCL als de betreffende tabellen veranderd zijn'
(**CHEMIE_VERANDERD** or **MED_VERANDERD**) and **CHK_LAGE_KALIUM_SPIEGEL_VEREIST_KCL**

CHK_LAGE_KALIUM_SPIEGEL_VEREIST_KCL: 'Bij een lage kalium bloedspiegel moet Kaliumchoride verstrekt worden'
LAGE_KALIUM_SPIEGEL and not(**KCL**)
THEN DO AddErrorMessage('Bij een lage kalium bloedspiegel moet Kaliumchoride verstrekt worden',
LA_KA_EN_GE_KCL)
ELSE DO AddSolvedErrorCheck(LA_KA_EN_GE_KCL)

DIGOXINE_GAAT_NIET_SAMEN_MET_LAGE_KALIUM_SPIEGEL: 'Controleer CHK_DIGOXINE_GAAT_NIET_SAMEN_MET_LAGE_KALIUM_SPIEGEL als de betreffende tabellen veranderd zijn'
(**CHEMIE_VERANDERD or MED_VERANDERD**) and
CHK_DIGOXINE_GAAT_NIET_SAMEN_MET_LAGE_KALIUM_SPIEGEL

CHK_DIGOXINE_GAAT_NIET_SAMEN_MET_LAGE_KALIUM_SPIEGEL: 'Bij een lage Kalium bloedspiegel mag geen Digoxine gebruikt worden'
LAGE_KALIUM_SPIEGEL and **DIGOXINE**
THEN DO AddErrorMessage('Bij een lage Kalium bloedspiegel mag geen Digoxine gebruikt worden',
LA_KA_EN_DI)
ELSE DO AddSolvedErrorCheck(LA_KA_EN_DI)

HOGE_UREUM_SPIEGEL_VEREIST_GENTAMYCINEONDERZOEK: 'Controleer CHK_HOGE_UREUM_SPIEGEL_VEREIST_GENTAMYCINEONDERZOEK als de betreffende tabellen veranderd zijn'
(**CHEMIE_VERANDERD** or **ON_VERANDERD**) and
CHK_HOGE_UREUM_SPIEGEL_VEREIST_GENTAMYCINEONDERZOEK

CHK_HOGE_UREUM_SPIEGEL_VEREIST_GENTAMYCINEONDERZOEK: 'patient heeft hoge ureum en geen onderzoek voor gentamycinespiegel'
HOGE_UREUM and not(**GENTAMYCINE_TEST**)
THEN DO AddErrorMessage('Patient heeft hoge ureum en geen onderzoek voor gentamycinespiegel',
HO_UR_EN_GE_SP)
ELSE DO AddSolvedErrorCheck(HO_UR_EN_GE_SP)

{**************** *CATEGORIE: ALLERGIEEN PATIENT* ********************************}
PATIENT_IS_ALLERGISCH_VOOR_AMOXICILLINE: 'Controleer CHK_PATIENT_IS_ALLERGISCH_VOOR_AMOXICILLINE als de betreffende tabellen veranderd zijn'
(**MED_VERANDERD** or **ALL_VERANDERD**) and
CHK_PATIENT_IS_ALLERGISCH_VOOR_AMOXICILLINE

CHK_PATIENT_IS_ALLERGISCH_VOOR_AMOXICILLINE: 'Patient mag geen amoxicilline, wegens allergie'
ALL_AMOXICILLINE and **AMOXICILLINE**
THEN DO AddErrorMessage('Patient is allergisch voor amoxicilline', AL_VO_AM)
ELSE DO AddSolvedErrorCheck(AL_VO_AM)

PATIENT_IS_ALLERGISCH_VOOR_PENICILLINE: 'Controleer CHK_PATIENT_IS_ALLERGISCH_VOOR_PENICILLINE als de betreffende tabellen veranderd zijn'
(**MED_VERANDERD** or **ALL_VERANDERD**) and
CHK_PATIENT_IS_ALLERGISCH_VOOR_PENICILLINE

CHK_PATIENT_IS_ALLERGISCH_VOOR_PENICILLINE: 'Patient mag geen penicilline, wegens allergie'
ALL_PENICILLINE and **PENICILLINE**
THEN DO AddErrorMessage('Patient is allergisch voor penicilline', AL_VO_PE)
ELSE DO AddSolvedErrorCheck(AL_VO_PE)

{************************************ sub rules *****************************}
PATIENT_ONTSLAGEN: 'Patient is ontslagen'
BTEST PatientDischarged

PATIENT_OPGENOMEN: 'Patient is opgenomen'
not(**PATIENT_ONTSLAGEN**)

PATIENT_OVERLEDEN: 'Patient is overleden'
BTEST PatientDiseased

OPNAME_LANGER_DAN_3: 'Opname is langer dan 3 dagen'
BTEST GetLengthOfAdmission > 3

COMPLICATIE: 'Er is een complicatie opgetreden'
INFARCT or **COMA**

BEHANDELING: 'Er is een behandeling gaande'
ARTERIE_LIJN or **BALLONEREN** or **BEADEMING** or **CAVH_D** or **CVVH** or **IABP** or **SWAN_GANZ**

NORMAAL_OPNAMETIJD: 'Opname tijdstip is binnen kantooruren'
BTEST (GetAdmissionTime > 800) and (GetAdmissionTime < 1800)

WEEKEND_OPNAME: 'Opname is in weekend'

```
TEST
 dagnummer := GetAdmissionDay;
 case dagnummer of
   1: Opnamedag := zondag;
   2: Opnamedag := maandag;
   3: Opnamedag := dinsdag;
   4: Opnamedag := woensdag;
   5: Opnamedag := donderdag;
   6: Opnamedag := vrijdag;
   7: Opnamedag := zaterdag;
 end;
 if (OpnameDag = zaterdag) or (OpnameDag = zondag) then
   TEST := TR
 else TEST := FA;
ENDTEST
```

SPOED_OPNAME: 'Opname is spoed'
BTEST GetUrgencyStatus

{************************ *veranderingen van tabellen* *************************}
OPNAME_VERANDERD: 'De opnametabel is veranderd'
BTEST dataList.IsChanged(OPNAMETABEL)

BEH_VERANDERD: 'De behandelingstabel is veranderd'
BTEST dataList.IsChanged(OPNAMEBEHANDELINGTABEL)

COM_VERANDERD: 'De complicatietabel is veranderd'
BTEST dataList.IsChanged(OPNAMECOMPLICATIETABEL)

CHEMIE_VERANDERD: 'De chemie tabel is veranderd'
BTEST dataList.IsChanged(OPNAMECHEMIETABEL)

MED_VERANDERD: 'De medicatie tabel is veranderd'
BTEST dataList.IsChanged(MEDICATIETABEL)

ALL_VERANDERD: 'De allergie tabel is veranderd'
BTEST dataList.IsChanged(OPNAMEALLERGIETABEL)

IND_VERANDERD: 'De indicatie tabel is veranderd'
BTEST dataList.IsChanged(OPNAMEINDICATIETABEL)

ON_VERANDERD: 'De onderzoek tabel is veranderd'
BTEST dataList.IsChanged(OPNAMEONDERZOEKTABEL)

{***************** *Mogelijke indicaties* ***********************************}
INFARCT: 'Indicatie is een infarct'
BTEST GetIndicationStatus('Acuut Myocard Infarct')

{***************** *Mogelijke complicaties* ***********************************}
COMA: 'Complicatie is coma'
BTEST GetComplicationStatus('Coma')

{****************** *Mogelijke behandelingen* *********************************}
ARTERIE_LIJN: 'Behandeling is Arterie lijn'
BTEST (GetTreatmentStatus('Arterie lijn op ICU')) or (GetTreatmentStatus('Arterie lijn op OK'))

BALLONEREN: 'Behandeling is ballonneren'
BTEST GetTreatmentStatus('Ballonneren')

BEADEMING: 'Behandeling is beademing'
BTEST GetTreatmentStatus('Beademing')

CAVH_D: 'Behandeling is CAVH(D)'
BTEST GetTreatmentStatus('CAVH(D)')

CVVH: 'Behandeling is CVVH'
BTEST GetTreatmentStatus('CVVH')

IABP: 'Behandeling is IABP'
BTEST (GetTreatmentStatus('IABP in op ICU')) or (GetTreatmentStatus('IABP in op OK'))

SWAN_GANZ: 'Behandeling is Swan ganz'
BTEST (GetTreatmentStatus('Swan Ganz op ICU')) or (GetTreatmentStatus('Swan Ganz op OK'))

{************************ *Mogelijke onderzoeken* ******************************}
GENTAMYCINE_TEST: 'Er is een gentamycine test aangevraagd'
BTEST GetExaminationStatus('Gentamycine dal / top')

{********************* *Mogelijke lab testen* **********************************}
HOGE_ASAT: 'ASAT is hoger dan 100'
BTEST GetChemicalResult('ASAT', 200) > 100

LAGE_KALIUM_SPIEGEL: 'Kalium spiegel lager dan 3'
BTEST (GetChemicalResult('Kalium', 200) <> -1) and (GetChemicalResult('Kalium', 200) < 3)

HOGE_UREUM: 'Ureum is hoger dan 15'
BTEST GetChemicalResult('Ureum', 200) > 15

{******************* *Mogelijke medicaties* **********************************}
KCL: 'Medicatie is KaliumChloride'
BTEST (GetMedicationStatus('Kaliumchloride iv')) or (GetMedicationStatus('Kaliumchloride po'))

DIGOXINE: 'Medicatie is Digoxine'
BTEST GetMedicationStatus('Digoxine')

AMOXICILLINE: 'Medicatie is Amoxicilline'
BTEST GetMedicationStatus('Amoxicilline')

PENICILLINE: 'Medicatie is Penicilline'
BTEST GetMedicationStatus('Penicilline')

{************************** *Allergieen* **************************************}
ALL_PENICILLINE: 'Patient is allergisch voor penicilline'
BTEST GetAllergyStatus(PENCILLINE_ID)

ALL_AMOXICILLINE: 'Patient is allergisch voor amoxicilline'
BTEST GetAllergyStatus(AMOXILLINE_ID)

{************************ *STATE RULES**}
BEGIN: 'Analyseer mogelijke fouten'
STATE
INITIALLY TR

CHK_INT: 'Controleer database na een periode van tijd'
STATE

CHK_EVENT: 'Controleer database na een event'
STATE

CHK_DEMAND: 'Controleer database na een demand van ICIS'
STATE

ALLERGIE_CON: 'Controleer op allergien'
STATE
THEN GOAL: **PATIENT_IS_ALLERGISCH_VOOR_AMOXICILLINE,**
PATIENT_IS_ALLERGISCH_VOOR_PENICILLINE

MEDICATION_CON: 'Controleer op medicatie i.c.m toestand patient'
STATE
THEN GOAL: **DIGOXINE_GAAT_NIET_SAMEN_MET_LAGE_KALIUM_SPIEGEL,**
LAGE_KALIUM_SPIEGEL_VEREIST_KCL,
HOGE_UREUM_SPIEGEL_VEREIST_GENTAMYCINEONDERZOEK

MISC_CON: 'Controleer overige regels'
STATE
THEN GOAL: **ABNORMALE_OPNAMETIJD_IS_SPOED,**
GEEN_VERKLARING_VOOR_LANGE_OPNAMEDUUR,
PATIENT_ONTSLAGEN_MAAR_HEEFT_COMPLICATIE_OF_BEHANDELING,
HOGE_ASAT_IS_INFARCT

{************************** *TRIGGER RULES* **********************************}
INTERVAL_ENABLED: 'Aanvraag controle van de database op basis van een tijd-interval'
BTEST (context = interval)

EVENT_ENABLED: 'Aanvraag voor de controle van de database op basis van een event'
BTEST (context = event)

DEMAND_ENABLED: 'Aanvraag voor de controle van de database op basis van een demand'
BTEST (context = demand)

{*************************** PROCESS SECTION *******************************}
PROCESS

{Begin context}
ON **INTERVAL_ENABLED** FROM **BEGIN** TO **CHK_INT**
ON **EVENT_ENABLED** FROM **BEGIN** TO **CHK_EVENT**
ON **DEMAND_ENABLED** FROM **BEGIN** TO **CHK_DEMAND**

{Context: Bekijk database op tijd-basis}
ON **PATIENT_OPGENOMEN** FROM **CHK_INT** TO **MISC_CON MEDICATION_CON ALLERGIE_CON**
ON **PATIENT_ONTSLAGEN** FROM **CHK_INT** TO *

{Context: Bekijk database op event-basis}
ON **CHK_EVENT** FROM **CHK_EVENT** TO **ALLERGIE_CON**

{Context: Bekijk database op demand-basis}
ON **CHK_DEMAND** FROM **CHK_DEMAND** TO **MISC_CON MEDICATION_CON ALLERGIE_CON**

{Context: Bekijk medicatie i.c.m toestand patient}
ON **MEDICATION_CON** FROM **MEDICATION_CON** TO *

{Context: Bekijk allergieen}
ON **ALLERGIE_CON** FROM **ALLERGIE_CON** TO *

{Context: Bekijk overige regels}
ON **MISC_CON** FROM **MISC_CON** TO *

APPENDIX B: Developed Delphi components

TData component

```
type
 TData = class(TComponent)
 private
  queryDatabase: TDatabase; {database}
  FDatabaseName: TFilename; {field for database name property}
  FAliasName: TSymbolStr; {field for alias name property}
  FKeepConnection: boolean; {field for keep connection property}
  FLoginPrompt: boolean; {field for login prompt property}

 protected
  procedure SetDatabaseName(ADatabaseName: TFilename); {set the name of the database}
  procedure SetAliasName(AAliasName: TSymbolStr); {set the alias name of the database}
  procedure SetKeepConnection(AKeepConnection: boolean); {set the keep connection property}
  procedure SetLoginPrompt(ASetLoginPrompt: boolean); {set the keep login prompt property}

 public
  constructor Create(AOwner: TComponent); override;

 published
   property databaseName: TFilename read FDatabaseName write SetDatabaseName; {the name of the
   database}
   property aliasName: TSymbolStr read FAliasName write SetAliasname; {the alias name of the
   database}
   property keepConnection: boolean read FKeepConnection write SetKeepConnection default
   FALSE; {keep connection option of the database}
   property loginPrompt: boolean read FLoginPrompt write SetLoginPrompt default FALSE; {login
   prompt option of database}

 end;
```

TRuleData component

```
type
 TRuleData = class(TData)
 private
  procedure SetDatabaseName(ADatabaseName: TFilename);

public
  regelInfo: TICISQuery; {rule info query}
  constructor Create(AOwner: TComponent); override;
  procedure OpenQueries(AAdmissionNumber: integer); {open queries}
  procedure CloseQueries; {close queries}

 published
  property databaseName write SetDatabaseName; {the name of the database}
 end;
```

TICISData component

```
type
 TICISData = class(TData)
 private
  {ICIS database queries properties}
  Opname: TIcisQuery;
  Opnamecomplicatie: TIcisQuery;
  OpnameBehandeling: TIcisQuery;
  OpnameChemie: TIcisQuery;
  Medicatie: TIcisQuery;
  OpnameAllergie: TIcisQuery;
  OpnameIndicatie: TIcisQuery;
  Patient: TIcisQuery;
  OpnameOnderzoek: TIcisQuery;
  procedure SetDatabaseName(ADatabaseName: TFilename);

 public
  constructor Create(AOwner: TComponent); override;
  procedure OpenQueries(AAdmissionNumber: integer); {open queries}
  procedure CloseQueries; {close queries}
  function FindFirst(AQueryName: string; AAdmissionNumber: integer): boolean; {find first record
  with admission numer in query}
  function FindNext(AQueryName: string): boolean; {find first record with admission numer in
  query}
  function GetField(AQueryName, AField: string): TField; {get field in query}
  procedure Next(AQueryName: string); {go to next record in query}
  function EOF(AQueryName: string): boolean; {look if EOF is reached}
  function GetQuery(AQueryName: string): TICISQuery; {get adress of query}

 published
  property databaseName write SetDatabaseName; {the name of the database}

 end;
```

TICISQuerycomponent

```
type
 TIcisQuery = class(TQuery)
 private
  admissionNumber: integer; {current admissionNumber}
  FChanged: boolean; {Field for enabled property}

 public
  constructor Create(Owner: TComponent); override;

 published
  property changed: boolean read FChanged write FChanged default FALSE; {indicates if the table
  has been changed by ICIS}
  function FindFirst(currentAdmissionNumber: integer): boolean; {find next admission number in
  list}
  function FindNext: boolean; {find next admission number in list}
 end;
```

TDataList component

```
TDataList = class(TList)
 public
  destructor Destroy; override; {destructor}
  procedure AddDataListElement(AName: string); {add element to list}
  function GetElement(AName: string; AItem: integer): PAdmissionRecord; {get element from list}
  function AddElement(AName: string; AAdmissionRecord: PAdmissionRecord): integer; {add
  element to list}
  function IsChanged(AName: string): boolean; {look if data list is changed}
  procedure SetChange(AName: string; AChange: boolean); {indicate if the list is changed}
  procedure Reset; {resets the list without deleting it}
  function GetCount(AName: string): integer; {get the count of the list}
end;
```

TErrorList component

```
TErrorList = class
 private
  errorList: TList;
 public
  count: integer; {count in list}
  constructor Create; {constructor}
  destructor Destroy; {destructor}
  function Add(AErrorListElement: PErrorListElement): integer; {add error message to error list}
  function Get(AItems: integer): PErrorListElement; {get element from error list}
  procedure Reset; {reset error list}
end;
```

TInfer component

```
TInfer= class(TObject)
 private
  dataList: TDataList; {list, containing the ICIS data}
  errorList: TErrorList; {list with error messages}
  possiblesolvedErrorList: TErrorList; {list with possible solved errors}

  outputDir: string; {directory where simplexys writes the output files}

  {private variables}
  _errfile: text;          {error storage file}
  _time0, _time: longint;  {time keeping}
  _dumpfile: text;

  _busy:  array [1..MAX_NUMBER_OF_RULES] of boolean;
  _R, _S: array [1..MAX_NUMBER_OF_RULES] of bool;
  _history: array [1..MAX_NUMBER_OF_RULES] of longint;

  _savfile: file;
  _buf:     array [0 .. 1023] of byte;
  _bytcount: 0 .. 1024;
  _bitcount: 0 .. 8;

  function AskVal(s: str80): bool;
  procedure skipexpr;
  procedure dump_buf;
  procedure dump_time (t: longint);
  procedure dump_rule (R: bool);
  procedure open_o_savfile;
  procedure close_o_savfile;
  procedure dump (s: string);
  procedure fatal_inference_error (s: str80);
  procedure non_fatal_inference_error (typ: word; rule: word);
  function sys_time: longint;
  procedure init_time;
  procedure update_time;
  procedure show_value (rule: integer; value: bool);
  procedure show_progress (typ: word; rule: integer);
  procedure show_applied (s: str80; value: bool);
  procedure setrule (rule: integer; value: bool);
  procedure thelse (rule: integer; value: bool);
  function evalexp: bool;
  function thelseb (rule: integer): bool;
  function evalrule(rule: integer): bool;
  procedure up_STATErules;
  procedure up_MEMOrules;
  procedure print_header;
  procedure getFACTs;
  procedure FDOS(_i: word);
  function FTEST (_i: word): bool;
  function FHIS (_i, _j: word): boolean;
```

```
  {private query procedures}
  procedure AddErrorMessage(AErrorMsg: string; ARuleNumber: integer); {add error message to
  list}
  procedure AddSolvedErrorCheck(ARuleNumber: integer); {check if error is solved}
  function GetAdmissionTime: integer; {get admission time from database}
  function PatientDischarged: boolean; {Is the patient already discharged}
  function PatientDiseased: boolean; {is the patient deseased}
  function GetAdmissionDay: integer; {get admission day of week from database}
  function GetLengthOfAdmission: integer; {get length of admission}
  function GetUrgencyStatus: boolean; {get state of urgence of patient}
  function GetComplicationStatus(AComplication: string): boolean; {has the patient a complication}
  function GetTreatmentStatus(ATreatment: string): boolean; {has the patient a treatment}
  function GetChemicalResult(ALabTest: string; ATimeInterval: real): real; {get the chemical status
  of the patient, -1 if not present}
  function GetMedicationStatus(AMedicine: string): boolean; {get the used medicine of the patient}
  function GetAllergyStatus(AAllergyID: integer): boolean; {get the allergies of a patient}
  function GetIndicationStatus(AIndication: string): boolean; {get the indication of a patient}
  function GetExaminationStatus(AExamination: string): boolean; {Get status of examination}

 public
  enabled: boolean; {indicates if inference engine is running}
  context: integer; {indicates the context of the inference engine}
  constructor Create;
  destructor Destroy;
  function GetErrorList: TErrorList; {get adress of error list}
  function GetPossibleSolvedErrorList: TErrorList; {get adress of solved error list}
  function GetDataList: TDataList; {get data list}
  procedure StartUp;
  function FirstRun: integer;
  function NextRun: integer;
  procedure CloseDown;

end;
```