Chapter 4 - Constructing the advice KB-DSS

The aim of this study is to provide a KB-DSS to students to select realistic relevant courses to support the completion of their current qualification in the most realistic time possible. The advice-giving knowledge component needs to be maintained by personnel that are course experts. A course expert is an expert on the relationships and rules that exist between the university's departments' study programs and courses. The first endeavour of this study will be to find a commercial software package or packages that will provide the desired functionality.

4.1 Selecting a suitable KB-DSS generator

A number of DSS and KB-DSS software packages were assessed to select an appropriate software package that allows domain experts to maintain a knowledge base of their knowledge. In the selection process, the focus was placed on the maintenance and creation of the rules as part of the knowledge component.

4.2 Software packages evaluated

The choice of as KB-DSS generator depends on the availability of a suitable knowledge component. The search for a suitable software package was not exhaustive. The student registration advice system was prototyped and presented as a solution after a suitable KB-DSS was discovered. No formal training was received on any of the software packages. The DSS software packages assessed are shown in Table 1. All software packages evaluated were DSS, KB-DSS or DSS generators. Not all of them contained an expert subsystem in their knowledge component. The software packages are listed in the order of experimentation (Table 4-1 (p42)). A short assessment of each of the above software packages is given below.

4.2.1 Criterium Decision Plus

This product assists decision-making in the form of setting goals and sub-goals, but does not contain an inference engine or expert subsystem. This option was not explored further.

4.2.2 ERGO

ERGO provided a DSS software sample (executable) choosing a colour printer. The tool to build an own application was not included in the demo package. This option was not explored further.

Table 4-1 Software Vendors and Decision Support packages

Product	Company	Web site:
Criterium DecisionPlus v 2.0	Infoharvest Inc.	http://www.infoharvest.com/
ERGO	Arlington	http://www.arlingsoft.com/
Expert Choice for Win	Expert Choice Inc.	http://www.expertchoice.com/
Statistica	Stasoft	http://www.stasoft.com/
Descision Pro	Vanguard	http://www.vanguardsw.com/
KnowMan Designer SR3	Intellix	http://www.intellix.com/
Business Rule Studio	Rule Machines Corp.	http://www.RuleMachines.com/
Visual Rule Studio	Rule Machines Corp.	http://www.RuleMachines.com/
AgentOCX	The Haley Enterprise	http://www.Haley.com/
Jess	Distributed Computing Sys	http://herzberg.ca.sandia.gov/jess
ILOG JRules 3.0	ILOG	http://www.ilog.com

4.2.3 Expert Choice for Windows

Expert Choice is based on the Analytic Hierarchy Process (AHP), a methodology for decision-making. It provides users with the tools to construct decision frameworks from both routine and non-routine problems and ways to include value judgements in these decision frameworks. This framework is a hierarchy, used to organise all the relevant factors to solve a problem in a logical and systematic way, from the goal to the criteria to the sub-criteria and down to the alternatives of a decision. The user must define the problem and enter all the relevant issues into the hierarchy. Expert Choice does not include an intelligent or expert component. This option was not explored further.

4.2.4 Statistica

Statistica includes an extensive set of mathematical models to assist a manager in the statistical assessment of a company's data during the decision-making process. The software analyses problems of a more computational nature. The advice problem has a cognitive nature and needs software with a strong intelligent or expert component. Statistica was not explored further.

4.2.5 Decision Pro

Decision Pro performs a wide range of tasks required in business decision analysis. It provides expert system development to automate routine decisions. The models can execute in a standard Web browser. Decision Pro allows one to build models that include logical rules as well as mathematical formulae. The rule-based models use a set of Boolean rules rather than numeric equations to calculate results. It either returns a True or False result (Binary Models) or classifies persons, places or things (Classification Models).

Decision Pro builds a logic tree of rules to be applied and this determines how the user is prompted for information. Decision Pro decides when specific questions should be asked by using the rules defined. It applies logical look-ahead to ensure that only pertinent questions are presented to the user. When finished, it returns a True or False value indicating for example whether the candidate qualifies for the loan or not. The user can easily see how the final decision was reached by following logical values down the branches. Decision Pro asks as few questions as possible and ensures that all questions are pertinently based on answers to previous questions.

When the rules change, adjustments must be made to the decision tree structure. The structure of how the various nodes are linked may also change. Data can be exchanged with other applications by reading and writing a common data file. One can only read and write ASCII text files. There are no facilities for processing binary files. As a decision tree must be updated, this option was not further explored.

4.2.6 KnowMan Designer SR3

KnowMan Designer helps the user to collect, store and distribute knowledge. This knowledge can be collected from any expert and is stored in files called knowledge domains. Knowledge obtained from an expert can be formalised into a working knowledge domain and made accessible to others. The end users gain access to the knowledge domain by means of one of Intellix' viewer products.

When using the viewer products, a single result is derived from the input data provided. No reasons for the result provided are given. When additional knowledge needs to be added, it is recommended that a new knowledge domain be built. As the anticipated prototype will contain knowledge that needs to be altered and updated regularly, this option was not explored further.

4.2.7 Business Rule Studio (BRS)

BRS is a software tool used by corporate managers, enterprise experts and software developers who wish to define and manage business rules as a separate unit owned by management and corporate experts and deployed as part of the application software in co-operation with the software developers (Business Rule Studio 1998). These rules are stated using standard representations familiar to management such as a spreadsheet-like decision table. The business objects such as customer, order and product can tap into intelligence such as a credit limit specification according to the account history, time of the year, product demand and receivables supplied as rules. These rules make business more adaptable to changes and more responsive to customers. The rules and objects are mutually dependent on each other. The rule repository provides clear and concise business rule authoring, documentation and management without any of the unnecessary complexities of programmer syntax, compilation, installation and distribution.

BRS is not an expert system. It comprises of a Rule Author for capturing business policies in the form of objects and rules, a rule engine for executing these policies, a rule repository to manage and store the

objects and rules, and a repository administrator for managing developer privileges and application deployment (Business Rule Studio 1998). A business rule is the combination of a set of constraints, the resultant actions and a business statement. The business statement is the message displayed to the client application user when that rule's constraints have been met. The tutorial provided links to a Visual Basic client application.

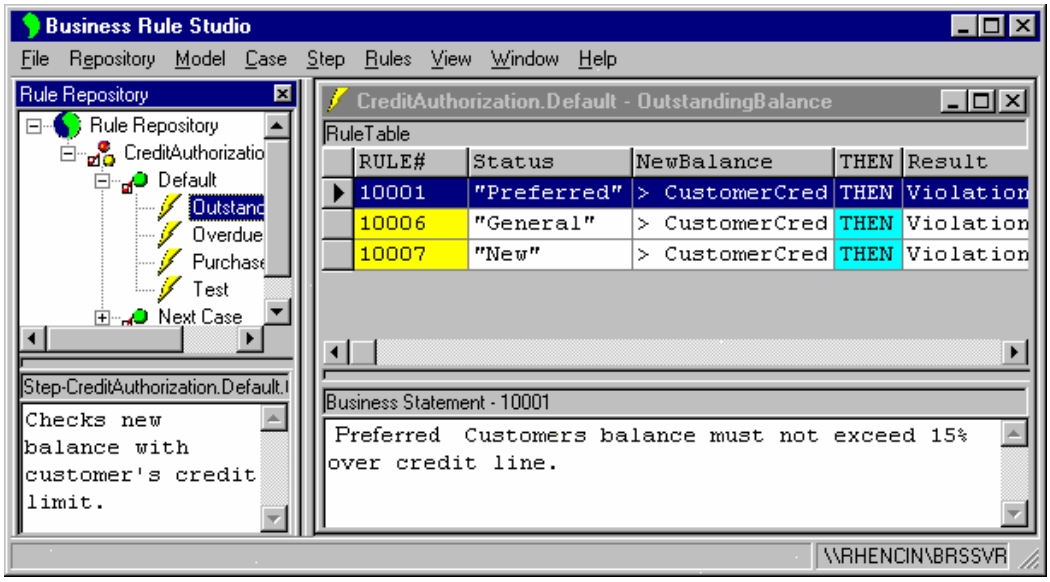

Figure 4-1 Developing the business rules (Business Rule Studio1998)

Management and enterprise experts can use the rule author component to create and maintain the rules within the designed core model structure established to interface with production software applications. The rule's interface in the form of a spreadsheet (See Figure 4-1) compares certain columns to specific values to determine cases when actions should be taken. The objective will be to set up a rule set with properties with relations between values of the same column to suite our case study. This seems cumbersome and may be due to the fact that BRS is not to an expert system. Thus, this software product seemed not to be a solution to the advice system.

4.2.8 Visual Rule Studio (VRS)

Another product from Rule Machines Corp., called Visual Rule Studio (VRS), may be a potential generator to develop expert systems, business rules and knowledge management applications. VRS might have a possible solution to our problem. The documentation states that VRS re-use the rules and knowledge in any client applications that support COM (Component Object Model) or OLE (Object Linking and Embedding) such as C, C++, Java, PowerBuilder, Delphi and many more.

The different sets of rules can be grouped together in a rule set. The rules are coded using VRS' Production Rule Language (PRL). It has an "if—then—" structure. The rule set is then compiled by VRS into a preferred type of executable: a standard exe a server DLL, an ActiveX control (OCX) or an ActiveX document. This compiled rule set is then referenced from within a client application such as

Visual Basic. The tutorial that accompanied the product demonstrates the functionality via a Visual Basic client application. According to the documentation, this can just as easily be done via the products as mentioned above as well as Internet or Intranet applications.

VRS uses backward chaining, forward chaining and hybrid or mixed chaining to solve complex business issues. The rule set can become part of the distributed exe application, or be distributed by a host, executing on a client or be centralised, running on a single host server supporting multiple client applications. VRS is a rule development environment for Windows NT, Windows 98 and 95 that may be deployed in an Internet/Intranet environment. VRS seems to be a viable option if the operating system used for the Advice Systems is one of the above-mentioned operating systems.

4.2.9 AgentOCX

♦ The Rete Algorithm

An inference engine that determines all applicable rules before applying them would intuitively perform linearly with the number of rules. The more rules that are added, the slower the inference engine would perform. The Rete algorithm was developed to provide a production system whose performance varied less than linearly. Given a significant number of rules, the Rete Algorithm performance is unaffected when adding more rules. There is no other algorithm, published or promised, that offers such performance (Agent OCX 1996).

♦ Functionality of AgentOCX

AgentOCX is an OLE control for use with Rapid Application Development (RAD) tools as well as conventional programming languages including C++. AgentOCX is a COM (Component Object Model) component that encapsulates the Eclipse Toolkit for Microsoft Windows 95 and NT within an OLE interface to embed Artificial Intelligence (AI) for business applications. AgentOCX provides rule-based programming not obtainable from other procedural or object-oriented programming tools. The Eclipse inference engine uses the Rete Algorithm that provides the only viable architecture for rule-based programming. Eclipse is among, the world's fastest inference engines and supports forward and backward chaining as well as other functionality, such as truth maintenance and case-based reasoning (CBR). Eclipse is available as a set of Dynamic Link Libraries (DLLs) for Windows 95 and NT. These DLLs export an Application Programming Interface (API) that allows an application to embed knowledge-based or expert systems implemented using Eclipse.

♦ Defining the rules using Eclipse

Unfortunately, the syntax specification of the rules in Eclipse is of such a nature that it will be difficult to be maintained by course experts. Haley Enterprise offers a rule editor called Authorete, but it is not available for evaluation purposes. AgentOCX together with Authorete seems to be worth exploring, but because the product must be purchased to investigate, this option was not explored further.

4.2.10 Jess

The Java Expert System Shell and scripting language (JESS) is compatible with all versions of Java starting with Java 1.1. Jess is an API library and is itself written in Java. Jess supports the development of rule-based Expert Systems, which can be tightly coupled to code written in the powerful, portable Java language (Friedman-Hill 1997). The Jess language is very similar to the language defined by the CLIPS expert system shell, which in turn is a highly specialised form of LISP. Many simple CLIPS programs will also run unchanged in Jess.

Jess can be used as a rule engine, which is a program that very efficiently applies a set of if-then statements (*rules*) to a set of data (the *knowledge base*) (Friedman-Hill 1997). Jess rules have a similar syntax to CLIPS for example:

```
(defrule DegreeComputerScienceDetermines
    (currDegree (degreeCode “02130002”) (studyProgram” 02133221”)
=>
    (assert (CourseDetails      (courseCredits 225)
                                (max100LevelCredits 70)
                                (min300_400LevelCredits 56)
                                (maxCreditsOtherDept 22)
                                (maxCreditsPerYear 56))
```

The documentation states that Jess' syntax is identical to the syntax used by CLIPS. The above rule may be translated into pseudo-English as follows:

DegreeComputerScienceDetermines rule:
if
the student's current degree is and his study program is
then
in order to obtain the qualification the student must have at least 225 credits
of which 70 on level 100, at least 56 on level 300 and 400, 22 credits from
other departments, maximum 56 credits per year.

Jess maintains a collection of facts called a knowledge base. Similar to a relational database the facts must have a specific structure. In Jess, there are three kinds of facts: ordered facts, unordered facts, and “definstance” facts (Friedman-Hill 1997). Ordered facts are simply lists, where the first field (the head of the list) acts as a sort of category for the fact for example: (shopping-list eggs milk bread) and (person "Bob Smith" Male 35). A course example could be: (currDegree “02130002 “ “02133221”) where currDegree is the head of the list, 02130002 the degree code and 02133221 the code used for the study program e.g. for Computer Science. Ordered facts can be added to the knowledge base using the “assert” function. The “facts” command shows all the facts in the knowledge base and the “clear”

command clears the knowledge base. The "retract" function removes a single specific fact from the knowledge base. A course example would be:

Jess> (reset)
TRUE
Jess> (assert (currDegree "02130002" "02133221"))
<Fact-1>
Jess> (facts)
f-0 (initial-fact)
f-1 (currDegree "02130002" "02133221")
For a total of 2 facts.
Jess> (retract 1)
TRUE
Jess> (facts)
f-0 (initial-fact)
For a total of 1 facts.

Unordered facts offer functionality similar to Java or JavaBeans objects that encapsulates fields. The fields are referred as slots (Friedman-Hill 1997). A course example could be: (currDegree (degreeCode "02130002") (studyProgram "02133221"). The "deftemplate" construct defines the slots. This functionality will be useful in the creation of the different types of rules and data involved in the advice process. A course "deftemplate" example:

```
(deftemplate currDegree "The student's current degree"
  (slot degreeCode)
  (slot studyProgram)
  (slot fme (type BOOLEAN))
  (slot studyYear (type INTEGER)),
```

would allow one to define fact like:

```
(assert      (currDegree (degreeCode "02130002")
              (studyProgram "02133221"))
```

When the knowledge base is reset, all the facts defined by the "deffacts" construct are asserted into the knowledge base, shortening a tedious process to assert all known facts.

The third type of fact, definstance, adds a specific Java or JavaBeans object to the knowledge base. The "defrule" construct supplied by Jess contains all the rules to apply to the knowledge base. A Jess rule is in the form of a 'if... then" statement as in similar rule-based expert systems. If the Jess rule

engine is active, a Jess rule is executed whenever its if-part (the left-hand-side or LHS) is satisfied (Friedman-Hill 1997).

Jess uses the Rete Algorithm and both forward and backward chaining. The Rete Algorithm is an efficient method to reduce the number of iterations in a rule loop when performing pattern matching. Past test results across iterations of the rule loop are remembered and only new facts are tested against any rule LHSs. As a result, the computational complexity per iteration drops to something more linear in the size of the fact or knowledge base. Jess tries to determine whether a fact in the knowledge base, matching the pattern of the particular rule's LHS, exists. If it is found, the rule is placed in the agenda to be fired or executed. As the rule executes some facts in the knowledge base are altered in such a way that a other rules evaluate to being true or matched and are placed in the agenda to be fired. The contrary is also true, when the rule that is fired causes some rules on the agenda to evaluate to no longer pattern matched, the rule is automatically retracted. The above describes the process of forward chaining of inference.

In a backward chaining system, rules are still "if... then" statements, but the engine seeks steps to activate rules whose preconditions are not met. This behaviour is often called "goal seeking". Jess also supports backward chaining. To use backward chaining in Jess, you must first declare that certain fact templates will be backward chaining reactive using the "do-backward-chaining" function. When a rule matches a rule as backward chaining, the rule's LHS is rewritten as being in need to be matched.

Rules are uniquely identified by their names. Rules may contain wildcards, functions and variables. The variables need to be tested. Each variable may be submitted to any number of tests within the rule to be pattern bound with the knowledge base. Several of the most commonly used Jess functions are wrappers for methods in the jess.Rete class. Examples are run(), run(int), reset(), clear(), assert(Fact), retract(Fact), retract(int) and halt(). You can call these functions from Java when running an application or applet (Friedman-Hill 1997). Each individual jess.Rete object is an independent reasoning engine. A single program may include several independent engines (Friedman-Hill 1997). Jess can be used in a multi-threaded environment. The jess.Rete class internally synchronises itself using several synchronisation locks. The most important lock is a global lock on all rule left hand sides': only one assert or retract may be processing in a given jess.Rete object at a time. This restriction is likely to be relaxed in the future. The rules used by Java embedding Jess are saved in a CLIPS file. The syntax of the rules is of such specialised format that an editor-compiler should guard the correctness of the rules.

4.2.11 JRules

ILOG JRules is a general-purpose expert-system generator that combines rule-based techniques and object oriented programming to help one to add rule-based modules to one's business applications (ILOG 2000 (a)). The JRules source code is implemented in Java and works in a completely object-oriented manner. ILOG JRules directly infers from the Java objects of the application without any

duplication. The design of the application and the Java classes are independent of whether you use ILOG JRules or not. JRules uses the Rete Algorithm (See Paragraph 4.2.9: p45) to perform pattern matching. The JRules engine can be used in all types of Java applications including applets running inside a browser (ILOG 2000 (a)).

Rules can be added to or be removed from the inference engine whenever needed. The rules can be added from numerous sources such as text files, strings or a stream from a URL (Uniform Resource Link). The rules are Java objects themselves that can be embedded in an application and can be manipulated. Java objects can integrate with ILOG JRules using the class ILRContext (ILOG 2000 (a)).

ILOG JRules has the following elements:

- A language
- An API (application programming interface – similar to Java)
- An inference engine
- A rules editor called the ILOG JRules builder with debugger and tracer, and
- Additional tools

ILOG JRules is written for Java developers and assumes that the application developers are familiar with the environment in which the rules are used. The rules themselves may however, be created and maintained by course experts using the rules editor or builder.

♦ ILOG JRules language, API and inference engine

The rules are written using a language of specific syntax and structure (ILOG 2000 (a)). The rules can be created and debugged using the rules editor or builder. The rule base is implemented in Java code (ILOG 2000 (a)) and is in the form of API's. The API's include the ILOG JRules libraries and inference engine used by the application and the rules in the rule base. The inference engine manages the interaction between the application and the rule base (20: p xiii) and uses the Rete Algorithm (ILOG 2000 (a)) mentioned in Paragraph 4.2.9 (p45).

♦ The ILOG JRules builder, additional tools and documentation

This graphical interface provides debugging and tracing capabilities. It consists of a GUI interface that enables the user to input rules in a simple and direct manner (See Figure 4-3: p51). When debugging the state of the inference engine, the Work Area (WA) or Memory (WM) and the contents of the agenda (rules enabled to fire) can be viewed (See Figure 4-2: p51).

The following additional tools exist:

- A faster application may be obtained by compiling the ASCII rules file into Java source code. Benchmark tests done on sample cases accompany ILOG JRules.
- A documentation generator similar to Javadoc facilities are provided

- Rules may access any JDBC compliant database using the relational database connectivity API, and
- A syntax checker may be invoked to find errors before attempting to execute the rules in an application. This part is particular useful for the case study (See Paragraph 4.4: p53) where the end users are not ILOG JRules programmers. Invoking the syntax editor will assist course experts in detecting errors when entering and maintaining rules.

A user's manual (ILOG 2000 (a)) is supplied with ILOG JRules. In addition the following documentation is provided:

- A reference manual of the ILOG JRules API's in Javadoc format
- A language reference manual, documenting the ILOG JRules language: Programmers familiar with the Java environment may create an ASCII rules file direct without using the ILOG JRules builder using this manual
- A builder's user guide, describing the builder GUI (Graphical User Interface) environment and syntactic editor for course expert rule builders
- A business rule language support, describing how to develop your own business language to builders to develop ILOG JRules rules to assist course expert rule builders/ administrators, and
- A business action language guide, which presents a sample language and describe how to extend and customise it

JRules seems to be adequate to specify a working set of rules. The builders provide functionality such as rule traces and the compiling of rules to assist the rule administrator or builder in maintaining rules in a simple and direct manner.

4.3 Comparison

In order to select an appropriate product (KB-DSS environment or generator), the following were set as criteria (See Figure 1-2: p3, for focus emphasis):

- The DSS or KB-DSS generator must include a knowledge component
- The knowledge component must include all relevant knowledge in a manner that course experts can maintain it
- Changes to the knowledge component should preferably not result in a new version of the KB-DSS/SDSS software application
- Possibility of deploying the software application on the Internet/Intranet will be an advantage, and
- The KB-DSS must be able to provide a reason for actions taken

All software packages evaluated were DSS, KB-DSS or DSS generators. Not all of them contained an expert subsystem in their knowledge component.

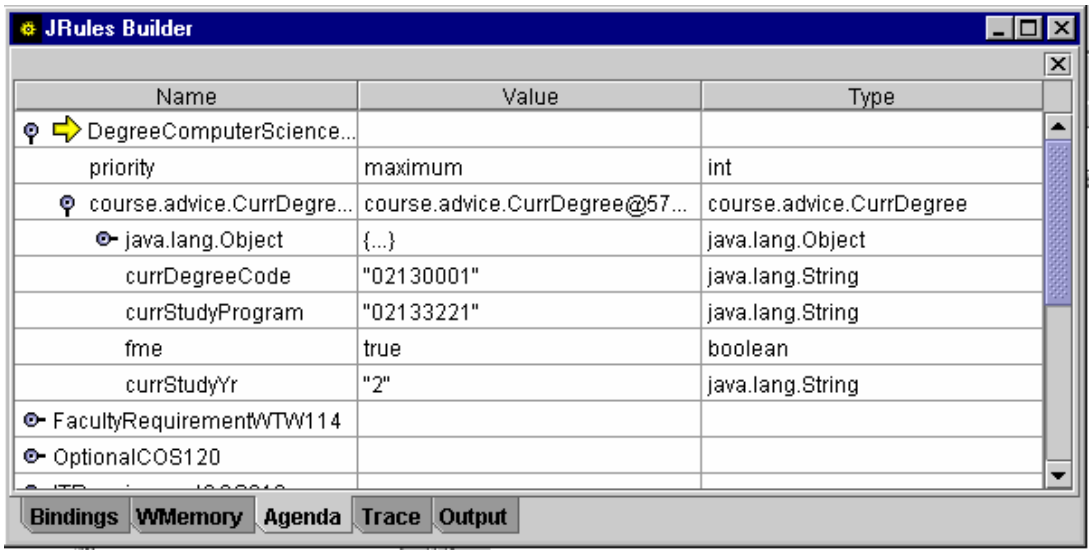

Figure 4-2 ILOG Builder's Agenda

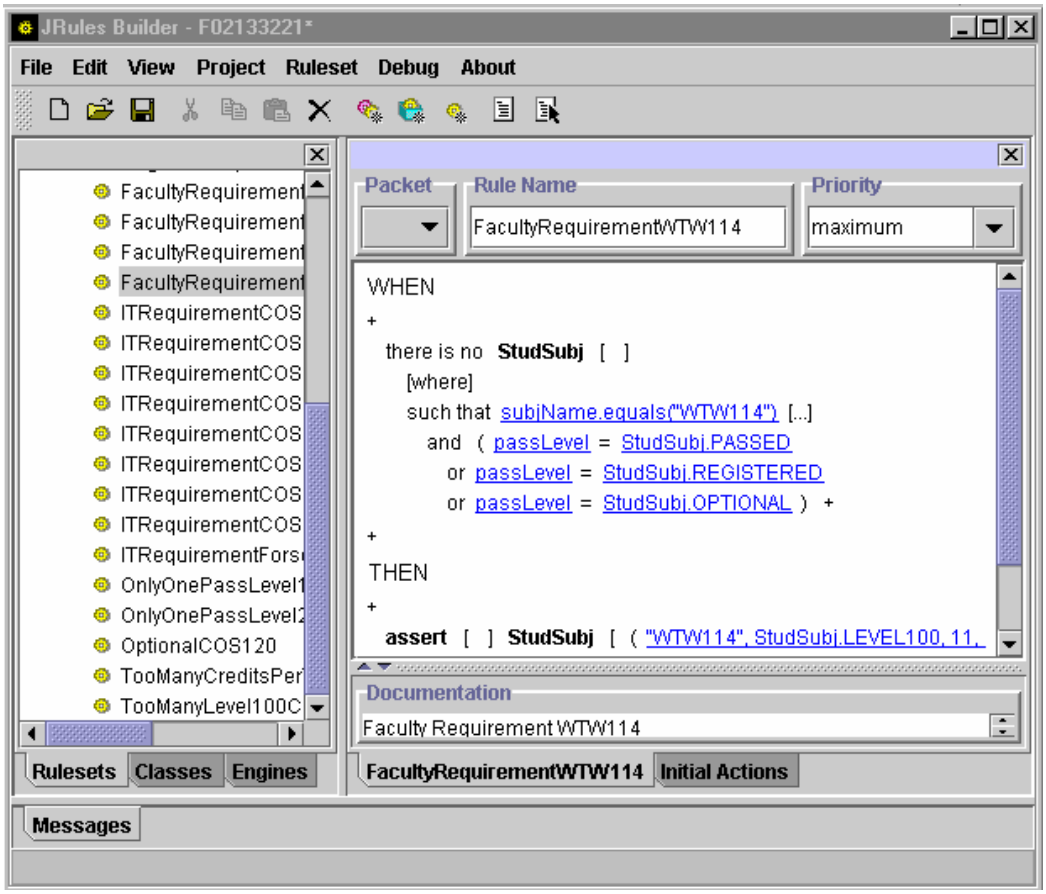

Figure 4-3 The ILOG Builder

Table 4-2 Summary of software selection criteria

Criteria / Software Package	Expert system, DSS or KB-DSS	Examples and suitable documentation provided	Demonstration version allows building of a course prototype	Reason for actions	Rules or knowledge maintainable by course experts	Release of knowledge changes	Web enable the application	Accept(A) or Reject (R) the software package
Criterium Decision Plus	DSS	Documentation, tutorial, examples	no knowledge base, setting goals and sub-goals in tree structure	tree, diagram	N/A	Rebuild goals and sub goals	N/A	R
ERGO	DSS	Example: choice of a printer, presentation	No	diagram	N/A	Recode	N/A	R
Expert choice for Windows	DSS	Documentation, tutorial, examples	goals and sub goals, tree structure	tree, diagram	N/A	Rebuild goals and sub-goals	N/A	R
Statistica	DSS	Documentation	Quantitative models, not assessed	N/A	N/A	N/A	N/A	R
Decision Pro	ES	Documentation, examples	Tree structure	tree	no	Tree structure change	yes	R
Knowman Designer SR3	ES	Documentation, examples	Knowledge domains	no	N/A	Knowledge domain change	no	R
Business Rule Studio (BRS)	DSS	Documentation, tutorials	No, spreadsheet with values. Not an ES.	N/A	Rules verification in the rules authoring tool	Business rule changes within the designed core model	yes	R
Visual Rule Studio (VRS)	KB-DSS	Documentation, tutorials for VB	Rules as ActiveX components used by a 4GL	yes, has a reason attribute	if..then structure in 4GL environment (No)	re-compilation necessary, re-deployment to application server: DLL creation	yes	R
Agent OCX	KB-DSS	Documentation, examples, tutorial	Embedded ECLIPSE from a 4GL	yes	Authorete™ *	Incremental compilation, use CLIPS files	yes	R
Jess	KB-DSS	Documentation, many examples provided	Yes, open source	yes	no specialised editor to check syntax or provide rule traces	Immediately: Replace CLIPS text file at URL	yes	R
ILOG JRules	KB-DSS	Documentation, extensive examples, tutorials provided	yes	yes	Various rule builders available: Platform specific	Immediately: Replace .ilr text file at URL	yes	A Rule builders specify rule builder platform

* Authorete™ supplied by Haley enterprises claims to include an easy maintainable rules component. Authorete was not available to be assessed.

The availability of a demonstration version, the ease of use of the package and the examples supplied with the demonstration version are included as criteria for a suitable DSS/DSS generator in Table 4-2. The search for a suitable product was not exhaustive. The primary reason a software package was rejected is presented in red. ILOG JRules was found to have the most potential according to the criteria set. A student registration KB-DSS was prototyped using ILOG JRules 3.0.

4.4 An advice system for student registration

At the University of Pretoria under-graduate students have a choice between various courses in the process of obtaining a specific qualification. Courses have unit counts, levels and prerequisites attributed to them. Qualification and course rules guard the course choices students have to make. These rules differ from department to department and change from time to time. Students find it difficult to choose the best course alternatives for the current semester. The department of Biological and Agricultural Sciences find themselves mostly occupied assisting students in their course selection. These enquiries are time consuming, leaving the experts in course content and prerequisites very little time for their other duties.

Giving students access to some form of automated advice system involving computer software would alleviate the university's departmental course experts from numerous identical queries to answer only those queries that need their expertise.

4.4.1 The need for a Decision Support System

Recall that a DSS in short is a computer-based system that aids the process of decision-making (Finlay 1994). In this sense, a Decision Support System could be a viable option for the student advice system. Although various educational institutions may experience this same functional problem, their departmental and course structures might differ completely from that of the University of Pretoria. The different departments at the university share a common database and therefore most of the structures used in the departments would correspond. The aim is to develop a custom-made DSS (See Paragraph 3.1.2: p32 and 3.1.3: p32) that would be shared among the different departments at the university. It would be of a great advantage if this system could provide an explanation for the advice it renders.

Not only would it enforce the system's credibility and trustworthiness with the users, but it would also train the users (students) so that they can assist one another or encourage the use of this system among their fellow students. If this DSS is available on the Intranet of the university, it will be available to all students without delay. Students would have the opportunity to do "what-if" analysis and so determine the optimum period of study. The system never tires and never becomes irritable and is never in a hurry. If necessary the same query can be submitted until full understanding is obtained.

4.4.2 An intelligent system's approach

The SDSS needs a component that presents the student (decision-maker) with all the possible options and a recommendation of the best courses to enrol for. This recommendation calls for an intelligent or

expert component as part of the DSS. Such a system is called a KB-DSS or ISS (See Paragraph 2.4: p23).

The different departments or fields of studies at the university have different sets of rules governing the process of course selection. These rules change often, causing the maintenance thereof using an automated conventional program almost incomprehensible, especially integrating the rules from all the different departments in one system. A solution is sought whereby departmental course experts could maintain their own rules separately (decentralised) from other departments as often as needed, suppressing the need to re-code and release a whole software application every time the rules change. Maintaining the rules should require little or no programming skills. A decentralised rule base would be a rule base stored on computers at multiple locations; however, a network does not interconnect the computers, so that domain experts at the various sites do not share rule bases (Adapted from decentralised database: Hoffer et al 2002).

An intelligent system reasons rather than computes a solution to a problem (See Chapter 6: p107 and Paragraph 2.4: p23). Knowledge or facts and experience are applied to the problem and one of many solutions concluded. The selection of the appropriate solution depends on the scenario and factors that surround the problem. Incomplete or uncertain data add to the complexity of the problem. In order to distinguish a suitable solution from an unrealistic solution, common sense may be applied. An intelligent system has a bit of this common sense included in its information bank and it may be referenced when necessary.

When a problem, such as the student advice scenario, has a restricted domain, the specifications of the problem can be communicated with ease. This expression of the specifications is an essential step in the intelligent system development (See Paragraph 3.1.4: p33). If such a system is of reasonable scope, it increases the probability of success to be reproduced by rules in a knowledge base (See Paragraph 6.1.1: p110). If a problem takes more than a few hours to be solved by a human, the mental processes and common sense used by humans to solve this problem would probably be overwhelming and impossible to be reproduced by a set of rules (Bielawski & Lewand 1991). Advising a student has a limited scope and should not involve hours of reasoning.

When the problem solution requires an explanation of how the system reached its conclusions, it confirms that an intelligent system's approach would be preferable (Bielawski & Lewand 1991). Advising a student involves reasoning and explanations. The focus of this study is to show that it is possible for a domain expert to supply and maintain a knowledge base of expert rules used in this Decision Support System for undergraduate students to obtain advice about the required and desired courses necessary in the current year of study. This dissertation focuses on the knowledge representation interface to be used by course experts that maintains rules used by a software application.

4.4.3 Requirements specification using a scenario-based approach

In order to derive a requirement's specification, a scenario-based approach was taken as specified by Haumer, Pohl & Weidenhaupt (1998). A current state analysis was performed to specify the functionality of the existing manual system. The Computer Science study program scenario, as outlined in the Rules and Syllabuses by the University of Pretoria (1999): Faculty of Science, was analysed as a use case to support the definition of the current-state model and to derive the change definition. According to Haumer et al (1998), the use of a scenario in the form of rich media, additional to the conceptual model, improves the quality of the requirements engineering process, leads to better understanding of the usage domain and enforces focused observation. Capturing the current system usage causes the abstraction process, which leads to the definition of the current-state model, to be more transparent and traceable, whilst explaining and illustrating the conceptual model. This process provides the basis to refine and detail the conceptual model during later phases.

Haumer et al (1998) focuses on the elicitation and validation of goals achieved by the current system and then outlines the interrelating goals and recorded observations. Their overall approach is shown in Figure 4-4 (p55).

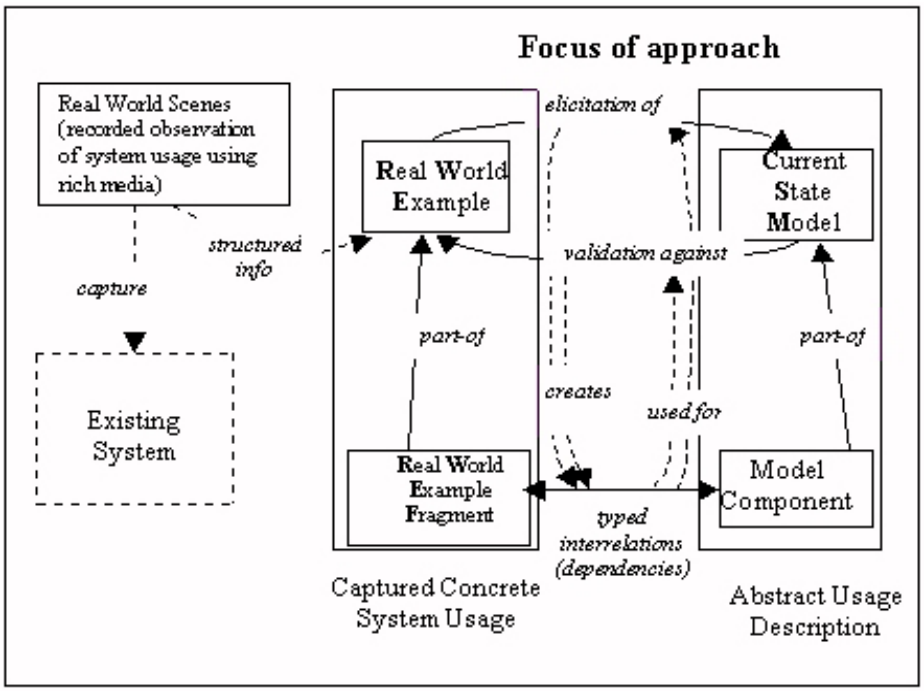

Figure 4-4 Interrelating Real World Example Fragments with model components (Haumer et al 1998)

Observations of the current system are captured and structured to reflect information about one system usage called a Real World Example (RWE). This approach uses dependency links to relate the goal to the parts of captured observations. Goals are chosen as the central concept for defining the current-state model. The RWE is used to validate the current-state model and to elicit new concepts. The fragment of the RWE that interrelates with the fragment of the conceptual model are elicited and

validated. This approach is suitable in cases where the functionality of the old system has to be duplicated in the new system.

The goal and sub-goals in the form of rules and requirements were specified. These rules and requirements were put together to form the initial formal requirements specification. The main goal was established as being the acquisition of a qualification and the sub-goals the acquiring of the necessary courses to complete a specific qualification in the shortest possible time schedule allowed.
The context of the custom DSS or SDSS is shown in Figure 4-5 (p56).

4.4.4 Derived problem definition

The rules and requirements that were derived as a result are summarised as follows:
To correctly advice the student, the student's results of courses attended needs to be considered. This should be an input to the advice system. According to a set of rules, possibly placed in a rules subsystem, some compulsory courses should be recommended to the student. A list of optional courses can be presented to the student to choose. At the end these choices should assist that the student complete his B-degree in the shortest period allowed. Some recommendations and alternatives should be presented to the student. The choices of the student should fit in with the timetable of the university – another input to the advice system.

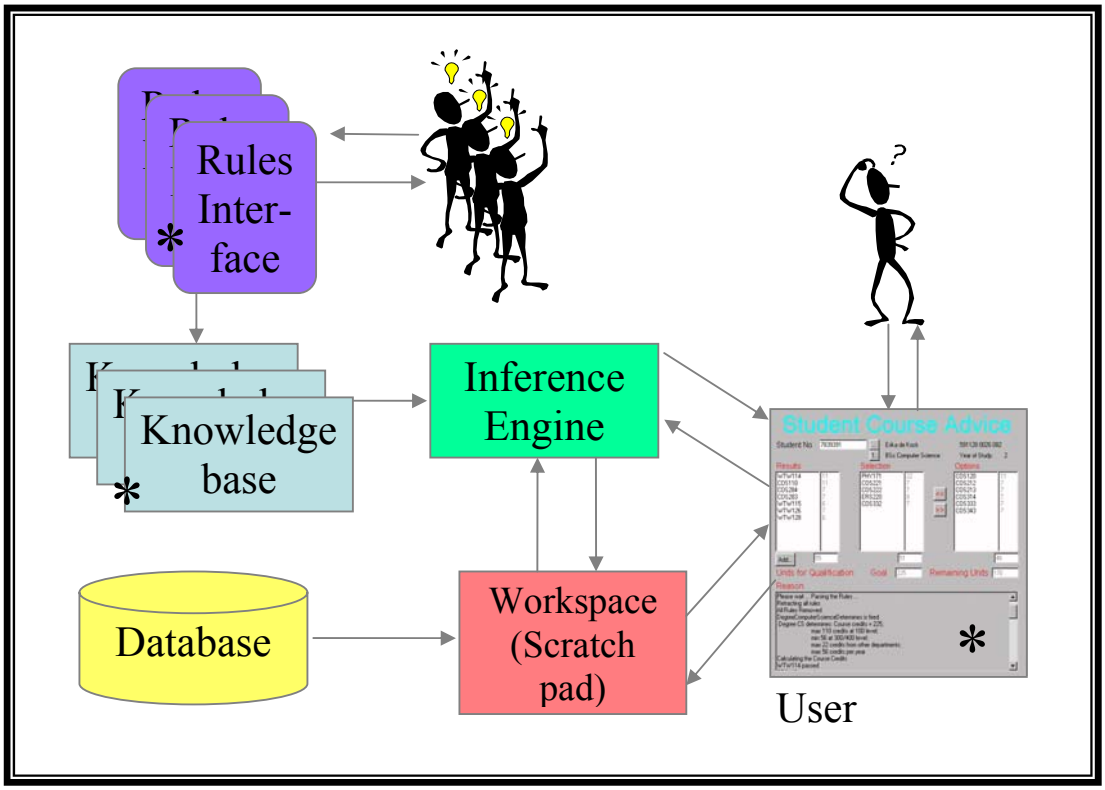

Figure 4-5 Context of the advice system

In this advice process, there should be components that:

- Recall the existing courses attended and passed by the specific student
- Specify the requirements of the specific degree and courses – possibly a set of separate rules for each study program
- Take these rules and infers advice to the student, matching the recommendations to the timetable of the university
- Present the advice to the student, and
- Offer a possibility to the course experts of the university to change the requirement rules

A summary of the rules derived from the scenario-based exercise using the Rules and Syllabuses of the Faculty of Science, University of Pretoria (1999) is as follows:
A student enrols for a specific B-degree. The degree has a name e.g. BSc, a unique code e.g. 02133221 and a study program name e.g. Computer Science associated with it. A degree consists of a number of courses that need to be included in the degree e.g. WTW114. Each of these courses has a number of credits assigned to it e.g. 11. Together with other course's credits, it will add up to a total number of credits needed to obtain the qualification. Every degree has a certain set of rules or prerequisites that needs to be adhered to, before a student can enrol (See Table 4-3).

Table 4-3 Pre-registration rule categories

Pre-registration rule categories:
Minimum performance of the prospective student obtained at subject level grade 12 for example:
– Mathematics HG: 50% or more, or SG: 60% or more
– Matriculation-score: 20 or more
University based rules e.g. obtaining a certain mark in a bridging course
Approval from the head of department or the dean of the faculty

Before a B-degree can be conferred on a candidate, a certain set of requirements should be adhered to (See Table 4-4). These requirements are categorised at degree level or at course level and may change in time. A separate component to maintain these requirement rules without prolonging the process before the rules are activated too much is thus a necessity.

Table 4-5 (p58) lists some of the types of rules specified for courses. Each course has a course code e.g. COS248. The first three characters are used to identify the subject e.g. Computer Science. The next character is used to show the level e.g. second year. This however is being invalidated by the anti-semester implementation. The last two characters are a unique number within the subject and the level to distinguish one course from another. These and other characteristics are shown in Table 4-6 (p58).

Whenever a student chooses a course that contradicts any of the rules, an explanation should be offered and the choice recalled. This study aims to investigate a possible solution to the above problem,

focusing on computerised support for decision making to benefit the students and the personnel of the university. Chapter 4 (p41) will focus on the construction of such a system.

Table 4-4 BSc Computer Science requirement rules

Degree requirements:
Minimum number of credits at final year level e.g. at least 56
Maximum number of credits at first year level e.g. maximum 110
Maximum number of credits from departments other than the home department of the field of studies e.g. 22
A student has to pass at least half of his courses per year
No third year subject done at other universities will be accredited
Minimum credits to confer on the degree e.g. at least 225

Table 4-5 Rules governing the courses

Rules governing the courses:
Compulsory courses as specified by the faculty
– Specific courses e.g. WTW114
– A subset of courses e.g. the following eight courses and at least four of a list of six courses
Recommended by
Degree e.g. six courses
Choice
Other

4.4.5 Prototyping the requirements

In order to discover and verify the requirements (Whitten, Bentley & Dittman 2001) of the above-mentioned problem, a user interface prototype was developed using Delphi. Prototyping improves user-developer communication and is more likely to meet user needs. This prototype is used as a surrogate specification to visualise specifications that would be written on paper, with the option that this prototype can evolve in into the finished product. The prototype approach being a surrogate specification is also known as throwaway prototyping. Throwaway prototyping is normally less suitable for Decision Support Systems (DSS), because of the different usage patterns of DSS. This factor makes the second approach: to evolve the prototype into the finished product better suited for DSS. This approach is also called evolutionary prototyping or rapid application development. The nearly finished appearance of the interface may however mislead users that the system itself is nearly done. In this case, much investigation and development needs to be done to produce an interfacing rules subsystem suitable to be maintained by course experts.

Table 4-6 Course characteristics

Each course has a

Course description

Level e.g. 1st, 2nd, 3rd

Home department e.g. Computer Science, Information Technology, Mathematics

Presenting department

No of credits e.g. 11

Semester or Anti-Semester: This would be 1st or 2nd semester or anti-semester 2nd semester but also presented in the first semester

Courses pre-requisites: The pass level could be one of the following:

–GS-A Final mark of at least 40% e.g. COS110 GS

–( )-Exam Entrance e.g. (COS110)

–....-Passede.g. COS110

–TD-Approval of the Dean

–TDH-Approval of Head of Department

–Excludede.g. should not be taken if COS110 was completed by the student

List of courses that should be taken in conjunction (passed or simultaneous) with a specific course e.g. COS110

Delphi was chosen as the prototyping medium, because of its rapid application development features and the possibility that the rules subsystem still to be discovered, may interface with it using Object Linking and Embedding (OLE) or ActiveX controls. The prototype was developed without a database interface and without the rules subsystem. It was intended to, if satisfactory; evolve into a further prototype including a rules or knowledge-based subsystem. The focus of the system is intelligent subsystem's interface, its knowledge base and its knowledge editor. The intelligent system's control mechanism is assumed a working component not to be designed by this study. The student registration user interface prototype is presented in Figure 4-6 and Figure 4-7.

The advice process starts by entering a valid student number. An interface to the database extracts the student's details as well as all the courses the student has passed. This is the input to the intelligent subsystem. The intelligent subsystem then suggests, by consulting its knowledge base and control mechanism, subjects to register for and other possible subjects to choose from (See Figure 4-7). The student can then enter a "what-if" scenario by adding subjects to his acquired list and selecting or deselecting subjects he wants to register. He applies his personal interests and goals for his studies and then the system responds using the intelligent subsystem approving or rejecting the student's choices and supplying the necessary reason(s) for it.

The information requirements of the advice system were determined in Paragraph 4.4 (p53) and an Information Requirement Definition (IRD) prototyped to understand the user needs and objectives of

the anticipated DSS/KB-DSS. Setting the objectives, understanding the user needs and formalising the problem is part of the first phase when designing a complex DSS (Turban 1995) (See Paragraph 3.1.6: p35).

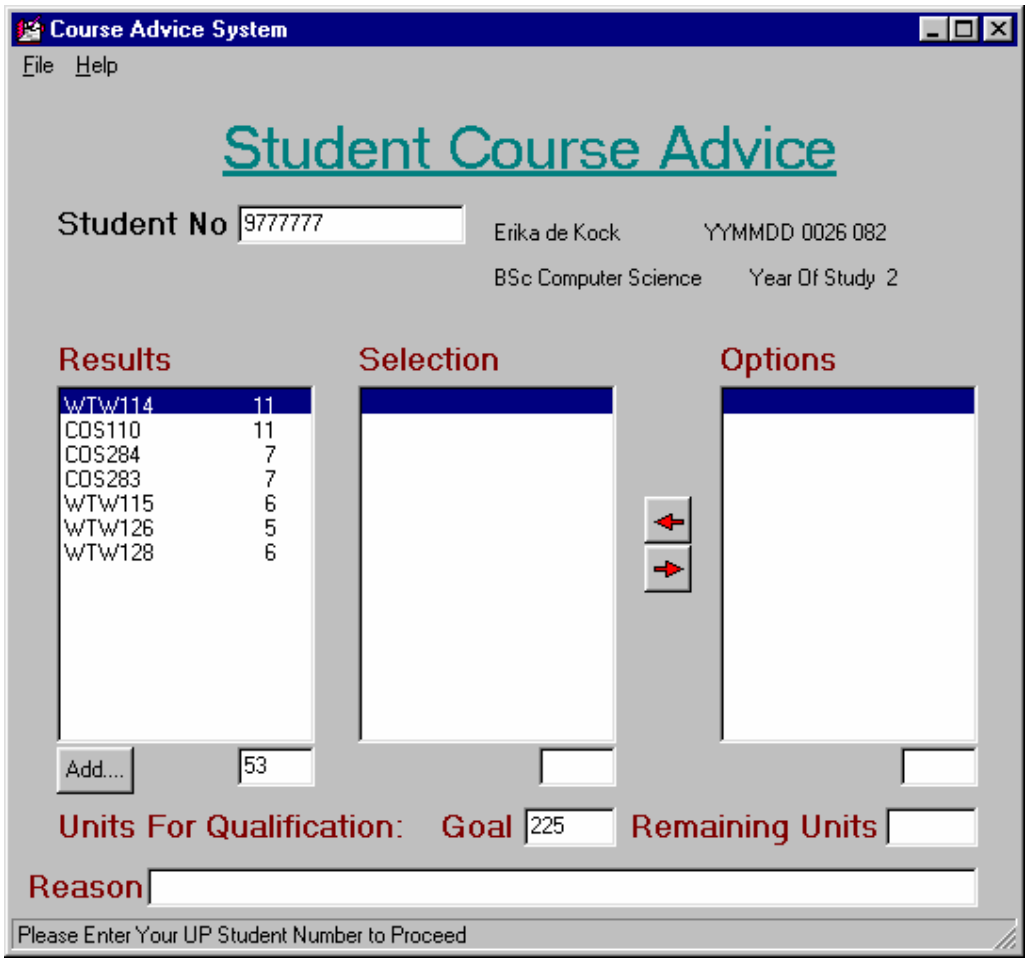

Figure 4-6 Initial prototype of the student advice system in Delphi

A prototyping approach called the evolutionary process (Keen 1980, as referenced by Turban 1995) or iterative process (Sprague & Carlson 1982) was adopted (See Paragraph 3.1.7: p35) for the remainder of the phases. Prototyping combines the four processes of the traditional SDLC (analysis, design, construction and implementation) into a single step that could be repeated several times until a relatively stable and comprehensible system evolves. Recall that the iterative process seems to be the most appropriate (See Paragraphs 5.1.4: p75 and 3.1: p32) to build a DSS. For this reason, the KB-DSS prototype will be developed from an end-users point of view (See Paragraph 3.1.7: p35). If no suitable KB-DSS packages exist, another construction process such as the implementation process by Klein & Methlie (1995) in Paragraph 3.2 (p37) may be followed. Klein & Methlie's (1995) methodology of implementing a KB-DSS will be used to assist Turban's (1995) development process (See Figure 3-3: p40 and Paragraph 3.2: p37).

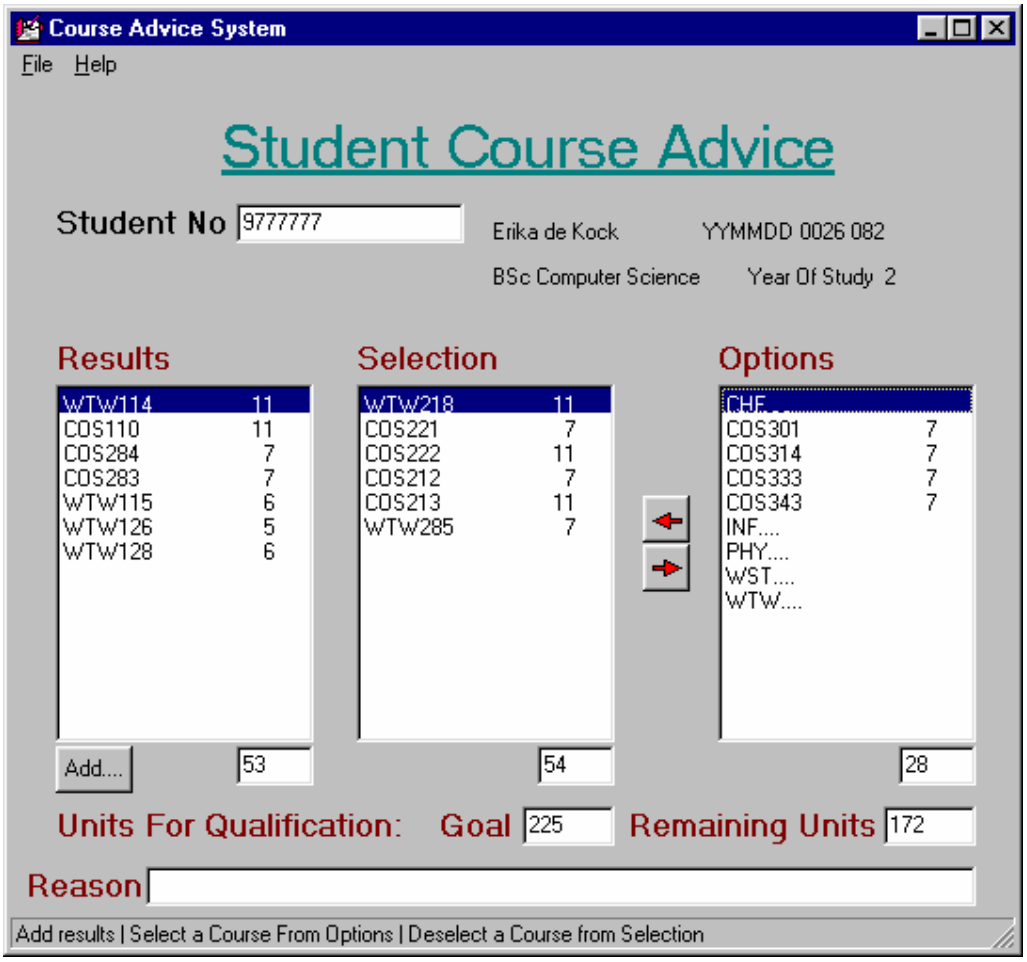

Figure 4-7 The user interface prototype after consulting the rules subsystem

4.5 The application prototyped

A prototype of the advice system was developed according to the examples shipped with ILOG JRules. In order to use the rule base in an application the ILOG libraries must be referenced in the coding. Commands such as retractAll( ) and fireRules( ) initiate the inference of rules within the prototype. Specific routines that realise as separate methods inside a particular rule class need to be specified. These routines manage the actions following the assertion and retraction of rules.

The prototyped application executes in two modes:

- Firing all suitable rules and displaying the explanations in a scrollable list box (… button), and
- Stepping mode firing one rule at a time and displaying the explanations rule by rule: At the same time the selection and options list boxes are updated with the recommendations (1… button)

The student may enter into a "what-if" scenario by selecting Options-list-box-courses to be Selected-list-box-courses. In the same way, students may choose to remove Selected-list-box-courses. In these actions, the students exercise their preferences. The expert component verifies the student's actions

and supplies reasons why the student may select or may not select such a combination of courses to complete the required qualification. The application gives the student the ability to view future semesters as well as allowing the student to add his passed courses. When pressing the Add… button, the student may enter codes of courses passed (See Figure 4-9: p63).

On exit, the explanation list box clears, the rules fire again, using the mode chosen and the updated Selection, Options and Reasons are displayed (See Figure 4-10: p64). Students may enrol for a certain unit count of courses of related departments. This functionality is not included in this prototype, but can be invoked by displaying the different departments in the options list box. On selection of the department, a list of the other department's subjects can be added to the Options list box to be selected by the student. Rules governing this selection process should be added to the current rule set or alternatively another rule set invoked to explain and allow selection of these courses.

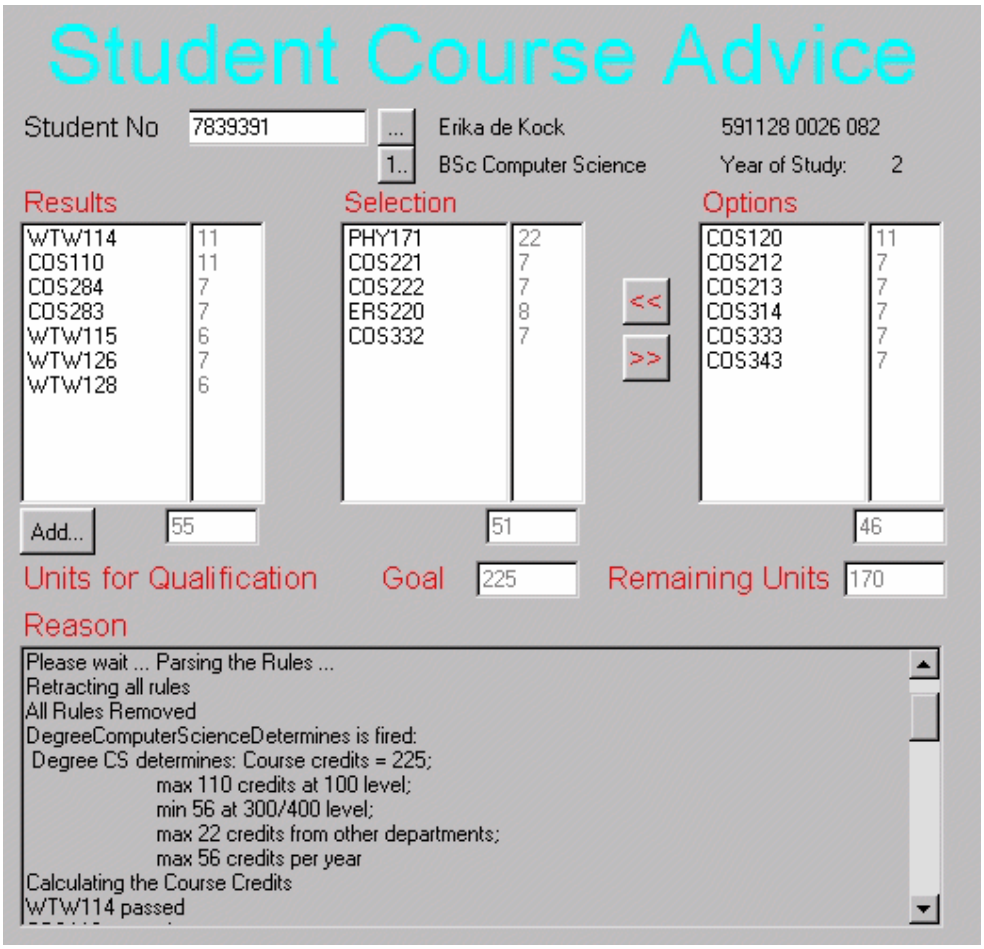

Figure 4-8 Working prototype of the advice system using ILOG JRules and Java

The student's final choice could be printed out, saved to a file, or e-mailed to him. A hard copy of the reasons could be provided as well. The student can enter into this "what-if" interaction as many times as he wishes. By matching his preferences to what is allowed by the rules he can make an optimum choice of courses for the next semester.

4.5.1 The rules prototyped using the ILOG JRules language

A prototype was developed using rules in the ILOG JRules language. The rules used by the prototype were set according to the specification for the system as presented in Paragraph 4.4.4 (p56) using the ILOG JRules language. An illustration of such a set of rules is given in Appendix A – Sample rules: F022331.irl. The Java source code for the prototype application is included on the CD. After a satisfactory rules file was developed using the ILOG JRules language and tested, the ILOG JRules builder was explored to determine if course experts would be able to maintain the rules.

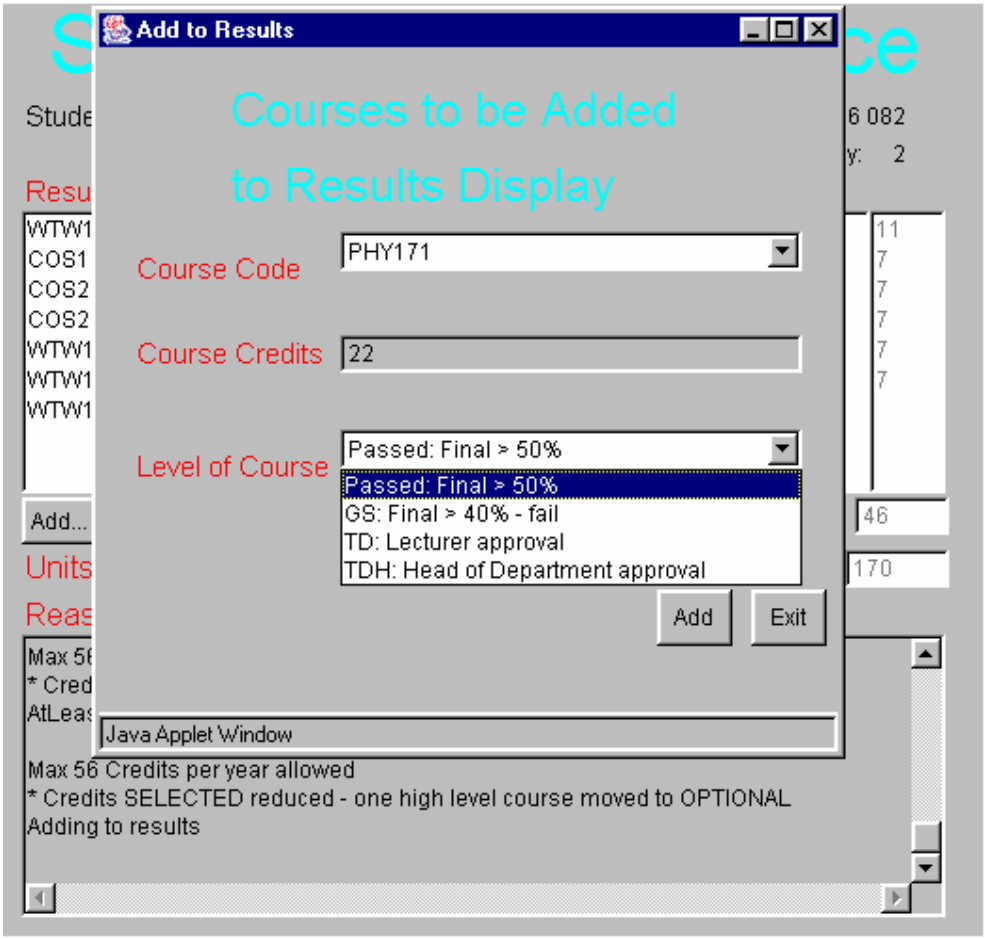

Figure 4-9 "What if" future semesters/years: Adding courses passed to results

4.5.2 The structure of a rule in the ILOG JRules language

An ILOG JRules rule has the following structure (ILOG 2000 (a)):

rule *myRule* {
    priority high;
    [packet = abc;]
    when { *conditions* …. }
    then { *actions* …… }
}

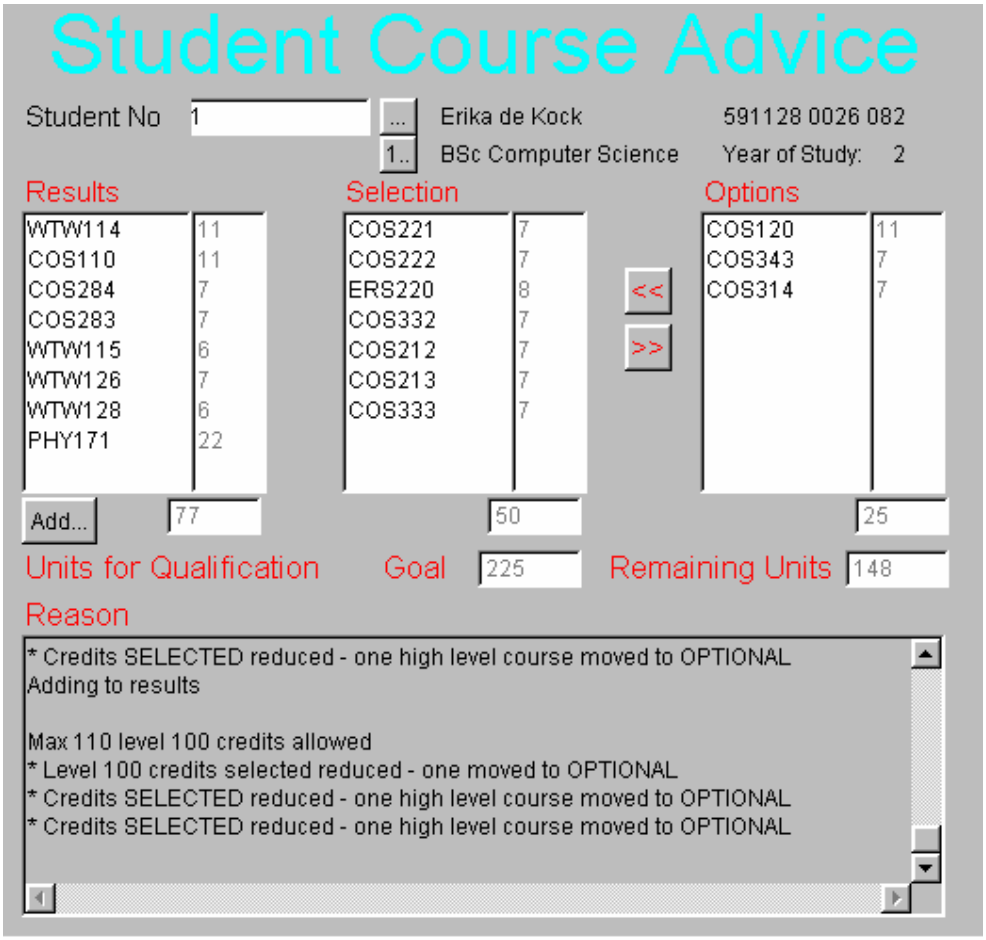

Figure 4-10 Selection and Options after new "what-if" results

Rules have unique names or headers (ILOG 2000 (a)), and are defined by the keyword rule: `rule DegreeComputerScienceDetermines {… }`. The rule has a priority, optional packet, conditions and actions. ILOG JRules infers its objects directly from the Java classes without duplication. Relevant objects are placed in the Working Memory (WM) by the assert command. The working memory is the place where ILOG JRules stores all its currently active objects. The agenda is a place where selected instances of rules are stored to be executed by the inference engine. A rule instance is a rule instantiated due to active objects from the working memory and is produced by any combination of objects in working memory that pattern matches the specified rule. A rule instance is a dynamic concept, whilst a rule is a static concept.

♦ Conditions

To determine a certain condition of a rule, the process of pattern matching is performed. The process determines the existence of one or more objects in working memory that matches the attribute values in the condition part of the rule. When a match is found an instance of the rule is placed in a place called the agenda. A negative pattern identified by a tilde (~) sign gives rise to a successful match if there are

no objects in the working memory that match the desired pattern (ILOG 2000 (a)). Whenever a successful match is performed, an instance of the rule is placed in the agenda.

More than one condition may be specified in a rule. Each condition must be on a separate line. All patterns of a rule must be matched before an instance of the rule is placed in the agenda. If one pattern is not successfully matched, then no rule instance is placed in the agenda and none of the actions of the rule performed (ILOG 2000 (a)). When a successful match occurs, the object may be marked with a variable (?x:) to be referred to in further patterns or in the actions of the rule. Each condition operates as a filter that selects a subset of objects. Variables may be used to reference a specific attribute value.

- **Actions**

Actions are operations that change the working memory. Two elementary actions are "assert" and "retract". Assert inserts an extra object into the working memory, while retract removes a particular object from the working memory (ILOG 2000 (a)).

- **Priority**

The priority part determines the order in which ILOG JRules' inference engine executes the different active rule instances. Two types of priorities exist: static and dynamic. Static priorities have specific values including a few predefined values such as maximum, high, low and minimum. Dynamic priority is used to alter the priority between instances of the same rule using variables defined in the condition part of the rule (ILOG 2000 (a)).

- **Packets**

Rules may be optionally grouped together using the keyword packet. A packet may be removed or added from the rule base and provides a means to activate and inactivate groups of rules (ILOG 2000 (a)).

- **Rule sets**

A rules set may contain both isolated rules and rules in packets. Each department will ideally have its own rule set of rules to maintain.

4.5.3 The inference process of ILOG JRules rules

Current objects are placed in the working memory, whilst matching rule instances are placed in the agenda. When an object is inserted into the working memory (action assert or initialise performed) the active rules are pattern-matched with all the objects and the relevant rule instances placed in a place called the agenda (ILOG 2000 (a)). The relevant rule(s) in the agenda are capable of being fired. To fire the rule(s), "fireRule( )" or "fireAllRules( )" has to be executed. These commands execute the action portions of the relevant rule instances and remove the fired rule instance from the agenda. The

contents of the working memory are changed and the satisfied rule instances placed in the agenda to be fired. Objects may be inserted into, removed from or updated in the working memory.

When several rule instances exist in the agenda, ILOG JRules' inference engine (ILOG 2000 (a)) has to decide which rule to fire using selection criteria such as priority, recency and order of rule names or lexicographic order. Firing a rule introduces a change to the working memory by means of actions (inserting, removing or updating objects) that might invalidate a rule instance that is already placed in the agenda but not yet fired. The invalidated rule instance automatically disappears from the agenda without firing it. This is also known as the truth maintenance system.

The working memory and the agenda evolve constantly and together these two containers constitute what is referred to as the ILOG JRules context. A context serves as an interface between the Java application and the ILOG JRules inference engine (ILOG 2000 (a)). A context associates a rule set with a working memory and implements the inference engine that controls them from the application.

4.5.4 The ILOG JRules Builder and syntactical editor

ILOG JRules assumes the rule programmer/builder is familiar with the environment in which ILOG JRules will be used. The Java Virtual Machine (JVM) needs to be installed together with the ILOG JRules software (ILOG 2000 (b)). A tool called ILOG JRules Builder exists that enables a user to create, modify and execute rules in a graphical environment. Rules created with this builder can be saved in a project file for later use. An environment called the syntactic rule editor allows both developers and users to write and edit rules in a more natural language, using menus to enter language statements and expression values.

♦ The ILOG JRules Builder

Rules may be written in different editor modes. Three rule editor modes exist (ILOG 2000 (b)):

- JRules syntactic mode
- JRules text mode, and
- Business Action Language syntactic mode

A Business Object Model (BOM) configures the rule editor and specifies the generation of executable rules to be used by an application. This is where an own business rule language may be specified. The builder provides a tracing and debugging tool where the contents of the agenda, the working memory, variable bindings, outputs and traces of the rules and active objects may be viewed. ILOG JRules may be launched in three different modes: offline, synchronised and advanced. When using the synchronised mode, the rules can be tested or debugged in conjunction with the application that uses it.

Upon entering the builder, a new project (.irp suffix) is created by default. The project is the root container for all objects and contains references to rule sets (.irs suffix) and BOM files (.bom suffix). Only one project may be edited at a time. One or more rule sets may be associated with the project. A

rule set will contain rules written according to a Business Object Model (BOM). A rules set contains a set of rules that may be grouped in packets. Rule files in the JRules syntax (.irl suffix) may also be loaded when opening a rule set (ILOG 2000 (b)). An example .irl file is included in Appendix A – Sample rules.

An empty BOM file is created when a new project is created. The Builder allows Java classes to be imported and configured in the BOM. Classes and class members may be hidden, removed or edited for use in the rule editor window. When a class is added to the BOM, all referenced classes are also added automatically marked as not visible not to be seen by the syntactic rule editor. The syntactic rule editor only uses the visible classes and class members. Setting a class or class member as not visible removes it from the possible choices in the rule syntactic editor. Whenever a class is removed, the builder removes all its subclasses and the members that reference this class. The column 'Display Name' may be edited to enter a name that will be used instead of the class element name with certain rule languages. The class list editor displays a filtered view of the BOM. Only visible classes and members are displayed. The syntactic editor uses only this filtered classes and members.

To add a new rule to a rule set using the rule Editor the rule language should be selected. Three choices are available (ILOG 2000 (b)):

- Text: the JRules standard language with textual view of the rule editor
- JRules: the JRules standard language with syntactic view of the rule editor, and
- Business Action Language: any custom-developed business rule language with the syntactic view of the rule editor: Any rules written in languages other than JRules need first to be transformed to JRules syntax before the rules are executable by the JRules engine.

The rule editor provides two views for editing: textual (See Figure 4-11) and syntactic (See Figure 4-12). The textual view is only available for rules written in the JRules language. The textual view offers the standard features of a text editor (ILOG 2000 (b)). At any time, the syntax of a single rule or the rule set may be checked. If the syntax is not correct, the rule error panel will list the errors. If the rule set does not contain any errors, it may be executed.

Execution includes several operations. It checks the rule set, create an engine or connect to an existing engine, send the rule set to the engine, reset the engine and fire all the rules (ILOG 2000 (b)). The rule set and all its rules may not be edited during execution.

Debugging commands exist to allow inspection of the working memory, the agenda and the variable bindings. The history of all events that occurred in the engine is accessible in the trace panel. The output panel displays the messages that have been sent to the output stream associated with the engine. Rule breakpoints, class breakpoints and object breakpoints may be set to evaluate all the relevant mentioned panels.

```
when  {
       ?x:CurrDegree(currDegreeCode.equals("02130001");currStudyProgram.equals("02133221"));
  }
 then   {
      assert CourseDetails(225,70,56,22,56);
  }
```

Figure 4-11 Using the textual mode of the syntactical editor

```
WHEN
+
      there is a CurrDegree [ ] called ?x +
            such that currDegreeCode.equals("02130001")
                  and currStudyProgram.equals("02133221") +
+
THEN
+
      assert [ ] CourseDetails ( 225, 70, 56, 22, 56 )
     [ so that ]
+
```

Figure 4-12 Using syntactic mode of the syntactical editor

After a satisfactory set of rules are compiled and tested the resultant rules file (.irs suffix) could be generated as a text file to be invoked as external rules by the application that uses the rules. The file data/F02133221.ilr in Appendix A – Sample rules is an example of such a file. The creation of this text file is the focus of this dissertation. This process of setting, editing and debugging the rules needs to be of a manner that course experts would be able to maintain their department's rule set with little or no assistance, taking ownership for the rules in the rule base.

Whenever a rule base is set and tested, it can be placed in a central place. This rule base then becomes the current active rule set as soon as the next enquiry by a student is made. The application need not even be terminated. Each department/faculty would have its own set of rules. The next enquiry by a student would query the database, obtain his department of faculty, derive the applicable rule file name to fuel the inference engine and invoke the newly created version of the rule file.

♦ Example rules from the advice system

In Figure 4-13 (p69), an example rule for a required course that has a pre-requisite, is given. The syntactical editor view of the rule is presented. Once the rule is created, it is easy to create similar course rules for the qualification. The course expert that is knowledgeable in courses and credits will recognise the course codes and credit values. The course expert knows that COS222 is an Information Technology requirement for the degree BSc Computer Science. The prerequisite COS110 has to be passed before the student can register for COS222. COS222 is presented in both semesters and the student can optionally choose to do COS222 in a later semester. COS222 is a second level course and contributes seven credits to the qualification.

```
Rule ITRequirement5
WHEN
+
      there is a StudSubj called ?x
            such that subjName.equals("COS110")
                  and passLevel = StudSubj.PASSED
+
      there is no StudSubj
            such that subjName.equals("COS222")
                  and ( passLevel = StudSubj.PASSED
                        or passLevel = StudSubj.REGISTERED
                        or passLevel = StudSubj.OPTIONAL )
THEN
+
      assert StudSubj ( "COS222", StudSubj.LEVEL200, 7,
StudSubj.REGISTERED, StudSubj.BOTH_SEMESTERS )

+
```

Figure 4-13 A rule from the student advice prototype for a required course that has a prerequisite course

A copy of such a rule can be changed to reflect the new course and its prerequisite e.g. "COS110" can be replaced by "ERS220"; "COS222" by "ERS320" and "7" credits by "8" credits. The new rule can be named CompulsoryCourseERS320 to include ERS320 and its prerequisite in the qualification for BSc Computer Science. The rule is frame-based (See Paragraph 6.2.2: p113). A frame called StudSubj exists with different slot values. The slot values can either be of a specific range of values or numeric and is created to support the rule.

If the pre-requisite COS110 is passed and the rules fired in the application, this course will appear in the Selection-list. The Selection list shows the recommended subjects the student should register for (Figure 4-8: p62). Explanations for the recommendations are given in the Reason-list box. "Information Technology Requirement COS222: Prerequisite COS110 passed" would be the explanation of the ES component for this course recommendation. The textual mode of the same rule is given in Figure 4-14 (p70).

Figure 4-15 (p70) shows a textual mode rule for a maximum of 100 credits allowed for level one courses. Slots should exist that contains the number of credits for a specific level. Another slot should contain the student's collective course credits for all passed and registered courses at a specific level. Before using additional frames and slots, e.g. CourseDetails and TestLevel100, for collections they need to be coded by a programmer as part of the application. Once the frames and slots are coded, course experts can maintain the rules.

```
When   {
      ?x: StudSubj(subjName.equals("COS110");(passLevel==PASSED));
      not StudSubj(subjName.equals("COS222");(passLevel==PASSED)
                        ||(passLevel==REGISTERED)||(passLevel==OPTIONAL));
 }
 then   {
assert StudSubj("COS222",LEVEL200,7,REGISTERED,BOTH_SEMESTERS);
 }
```

Figure 4-14 The textual mode of an example rule for the student advice system

```
When     {
        CourseDetails(?a:level100Max);
        ?c: collect ( new TestLevelSubj(LEVEL100))
        StudSubj(level==LEVEL100;(passLevel==PASSED)||(passLevel==REGISTERED);
                  ?y:noOfUnits)
        where (totalNoOfUnits() > ?a);
    }
   then      {
          assert(?c);
    }
```

Figure 4-15 Textual mode rule for a maximum number of 100 credits for level 100

♦ The syntactic rule editor

The syntactic rules editor of the builder allows the writing and editing of rules in a more natural language than using the textual editor. This editor allows non-JRules language programmers to write and edit rules. Menus provide a user-friendly way to enter statements and expression values. This editor may be used to specify rules in an in-house developed business rule language too. To develop an appropriate set of rules, the given sample language was found to be inadequate. The sample language needs to be extended before a set of usable rules can be created to assist the prototyped application. In particular, rules that combined objects into collections were difficult to specify using the sample language. The functionality to specify it using the JRules syntax existed.

To provide such a self-developed business rule language, a Business Object Model has to be defined. The development of such a language (business action language by specifying a business object model) may be the content of an additional study. Setting up this specification needs the expertise of programmers familiar to the JRules environment. This could provide an even more user-friendly way

to specify rules and expression values to be used by domain experts. The business language may also be used as an independent Java Bean (ILOG 2000 (b)).

A new Business Object Model is a viable solution, but outside the scope of this study. It may be a valid topic for further research in the area. The following discussion will focus on the editor and Business Object Model provided by JRules. The aim is to determine if the provided model and editor is sufficient to be used by the different departments to set up the rules needed to automate the students' enquiries on electing possible courses.

- **The textual mode**

The condition part of the JRules rule as discussed in Paragraph 0 (p63) is substituted with 'there is a' or 'there is no' followed by a selection of the applicable class. The 'where' clause incorporates fields of the class and binds them to local variables and 'such that' write tests on field values (ILOG 2000 (b)). When collections are evaluated in a condition, the token 'the selection of' may be used. By selecting the token '+', any number of tests may be incorporated in a single rule. The '+' token is also a delimiter between actions in the action part of the rule. Possible actions include 'assert', 'retract', 'update', 'modify', 'apply', 'bind', 'execute', 'if', 'while' and 'timeout'. The token 'so that' is used to assign values to the object fields of the selected class. Rules or portion of rules may be inserted, deleted, copied, cut and pasted to facilitate the specification of similar rules for example prerequisites for certain courses.

- **Business Action Language**

The business action language is a sample business rule language supplied with JRules, using a natural and readable syntax (ILOG 2000 (b)). Only a subset of the JRules language is implemented and is therefore limited in its application. These rules are translated into executable JRules rules before use in an application. This language can be used to define an own customer business action language by specifying a Business Object Model (BOM). As stated before, the definition of such a language is beyond the scope of this study. Mapping the BOM to the application classes can specify a whole vocabulary to be used by the rule administrators.

4.5.5 Evaluating the creation of rules using the rule editors

Objects can be grouped into collections and tested against certain conditions. These rules are the toughest to set and would need the assistance of a programmer/builder. Examples are TooManyCreditsPerYear, and TooManyLevel100CreditsMoveLastOneAdded. Appendix A contains sample rules in the different modes used by the advice SDSS prototype. The syntactic rules editor of the builder allows the writing and editing of rules in a more natural language than using the textual editor. This editor allows non-JRules language programmers to write and edit rules. Menus provide a user-friendly way to enter statements and expression values.

Adding similar rules and changing existing rules is quite easily achieved. Many of the rules will be of the same type e.g. prerequisite courses, total number of credits for a level, total number of credits for the qualification, etc. These rules could easily be maintained by changing the credit count, adjusting the prerequisites, adding new pre-requisites. The rule administrator, when needed, can create similar rules. The rule administrator would need to copy an existing named rule as a new named one and change the field names and values. The field values and the course codes can be changed to reflect the new prerequisite. If any changes need to be applied to existing rules, the field values can be replaced. If any of the rules are no longer valid, they can be removed from the rule set or package. Course expert rule administrators can maintain these types of rules using either the textual and syntactic modes of the builder.

When, however a new type of rule, different in structure to any other rule in the rule set, a builder/programmer's assistance will be needed. If a collection type rule, such as, "Only a certain amount of units for specific course level or attribute allowed", is needed, it will be safer to allow a builder/programmer expert to code or assist in coding such a rule. When altering the knowledge base, the rules need to be verified. Problems (See Paragraph 7.2.2: p130) that can be detected are:

- Consistency problems such as redundant rules, conflicting rules, subsumed rules and circular rules, and
- Completeness problems such as missing rules and gaps in the inference chains

These logical problems may not necessarily cause reasoning faults.

Using the debugging commands supplied with the builder environment can perform rule traces. It can also assist in performance validation. The debugging commands allow inspection of the working memory, the agenda and the variable bindings (ILOG 2000 (b)). The history of all events that occurred in the engine is accessible in the trace panel. The output panel displays the messages that have been sent to the output stream associated with the engine. Rule breakpoints, class breakpoints and object breakpoints may be set to evaluate all the relevant mentioned panels. The rule administrator needs to be trained in using this debugging environment to determine the validity of the rule base.

It is possible to set up the rules file (.ilr) using an ordinary text editor. The rules are typed and saved in an .irl file. The syntax of these instructions is checked on execution of the DSS and a system failure will result if this .ilr file is incorrect. Rule editing using an ordinary text editor is suitable for the programmer/builder only. The rule administrator can maintain the rules using the syntactic builder.

♦ Summary

A programmer/builder should set up the initial rules sets and any subsequent rules of a new type. Rule administrators can be allowed to maintain the rule set by adding similar rules using copy and paste and by changing existing field values of rules. The rule administrator needs to be trained to debug the rule base, verifying the completeness and consistency of the rules as described in Paragraph 7.2.2 (p130). Rules should be tested, validated and compiled before using it with the active SDSS. JRules has an

expert system as an integral component that can supply recommendations and explanations to the user, who is the student.

The knowledge component is in a limited sense maintainable by course experts. Course experts can easily copy, paste and change rules as discussed. Once trained in the use of the software, the interface would be logical and useful to the experts, especially when the copying of rules and the changing of field values are required to maintain the rule base. The knowledge component interfaces with the SDSS/KB-DSS application classes and is available to departmental experts. Departmental experts can access the rules decentralised or independent of the users and other experts. Once the rule base is verified, it can be released and copied to the JRules server and released into production. The JRules Builder 3.0 uses Remote Method Invocation (RMI) to communicate with the JRules engines.