CODEN: LUTFD2/(TFRT-5429)/1–52/(1990)

A Reasoning Machine Generator for Object-Oriented Programming Languages

Gustav Bergman
Christian Söderberg

Department of Automatic Control
Lund Institute of Technology
December 1990

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
Masters thesis

Date of issue
December 1990

Document Number
CODEN: LUTFD2/(TFRT-5429)/1-52/(1990)

Author(s)
Gustav Bergman, Christian Söderberg

Supervisor
Jan Eric Larsson

Sponsoring organisation

Title and subtitle
A Reasoning Machine Generator for Object-Oriented Programming Languages

Abstract

This thesis describes a system generating expert system-like systems, here called reasoning machines. The reasoning machines have been devised so that the rules could handle any number of objects, and that the backward chaining procedures will find all sets of objects satisfying a given goal.

In our thesis we devise a system, create methods to implement it, and finally create a cross-compiler for it. The implementation has been done in Simula, but could have been done in any object-oriented language — neither the system nor the methods are bound to Simula.

The final system consists of a cross-compiler, translating rule definitions into Simula classes, and an environment in which the generated code could be used. We have also, as an addendum, made some utilities to make the usage of the system more practical.

Key words
Backward chaining, Expert systems, Hard-coded rules, Inference, Multi-object reasoning machines, Simula

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
52

Recipient's notes

Security classification

Contents

1 Preface

This thesis can be divided into tree parts:

- The **design of a system** for multi-object reasoning machines – what this means will become evident later on
- The **construction of methods** for making such a system work properly
- The **implementation of a generator** for systems such as described above

These are quite different tasks, but they have been delt with simultaneously, i.e. when designing the system we took into account the difficulty we would get when solving the technical aspects of the system, and when we solved these technical difficulties we made certain the solutions were general enough to be implemented without too much difficulty. We want to emphasize that the design of the system was at least as challenging as was the construction of the methods. Indeed, the designing of the system required a lot more creativity than the other two tasks combined, even if this report may give an other impression.

Although, as indicated above, these three developments were made at the same time this report describes them one after another. In the first chapter we discuss some aspects of expert systems in general and then turn to the description of a kind of expert system-look-alike systems dealing with a multitude of objects at the same time. Chapter two describes how to solve the technical difficulties of such a multi-object system whereafter we in the third chapter briefly describe how to generate systems of this kind.

The methods devised are general enough to be implemented in any object-oriented language. Since we chose Simula for our implementation we have used Simula-syntax to describe ideas which could be described in any other language. We would naturally have preferred to use a standardized object-oriented description language to do this but, alas, to our knowledge, no such language exists.

We have written this thesis with the words of Ludwig Boltzman in our ears, leave elegance to the tailors and shoemakers. In numerous ways the ideas could have been more elegantly presented, but at the cost of clarity.

We would like to give a cordial thank to our distinguished supervisor, Jan Eric Larsson at the Department of Automatic Control, for the encouragement and constructive critisism which has helped us to complete this thesis.

We would also like to thank Göran Eriksson, DNA-LTH for supporting us with Lund Software House's Simula Compilator for MacIntosh MPW.

2 Introduction

2.1 Description of the problem

When the two of us, during a course in artifical intelligence, implemented a small expert system we used an expert system shell named Nexpert [Nexp88]. The shell was quite nice to use, with excellent graphical facilities and an *object oriented* language to describe things. The system had, however, some limitations.

In our project we had a number of objects we wanted to compare to each other. We did not always know beforehand how many objects we should investigate and Nexpert provided no simple way for us to *decide the number of objects at runtime.* Also the language supplied with the system, good as it may have been, did not have the necessary *calculational power* for some of our purposes.

The possibility of generating a *stand alone application* was another useful feature not provided by Nexpert.

We also thought it would be interesting if two or more reasoning machines could be used by the same program. In the AI terminology this kind of system is called an *Expert Committee.* We decided to try to make our system able to let the same program use several reasoning machines at the same time, although this was not a primary goal.

After a discussion with our supervisor these ideas were turned into a thesis, with the following goals:

Create a system with the following features:

1 The system allows the user to describe his objects in an object oriented manner.
2 The number of surveyed objects in the system can be determined at runtime.
3 The various objects in the system can be compared to, and combined with each other, in a simple way.
4 The system features powerful calculational abilities, at least as powerful as those found in an ordinary computing language.
5 The system give the user stand alone applications.
6 One application shold be able to use several different reasoning machines.

By "create a system" we mean to

- specify the syntax and semantics of such a system,
- develop general methods for making the system work, and
- write a program to automatically generate systems.

It was essential that these undertakings were done at a reasonable cost, to create a system which could seriously compete with Nexpert in all aspects is, of course, a task far too big to fit into a work of this

size. We therefore made the system so simple it could be without interfering with our main objectives.

In the beginning we did not know whether to make an interpreter or a generator. After some consideration we found that objectives 1, 4, and 5 above indicated a generator as the best solution.

Our choice of language for the implementation was Simula [Erik85] [Birt79], we had both used Simula earlier and Göran Eriksson had the kindness of providing us with a Simula-compiler to be run under MPW on Macintosh [West88]. It should be pointed out though, that the implementation of our methods could have been made in other object oriented languages, such as Smalltalk [Gold83], or C++ [Strou89] as well. The only prerequisite is that the language allows simple inheritance[1].

[1] As opposed to multiple inheritance, which is not required by our system.

3 Developing a multi object reasoning machine

In this chapter we will describe the fundamental aspects of the system we developed. We commence by discussing basic concepts of computer reasoning in general, and proceed to show some of the features we devised for our system. The chapter is concluded by a description of our final system specification.

3.1 Deduction, rules, facts and Modus Ponens

Human reasoning is a tremendously complex activity. When we reason we use several kinds of different methods, intuition, association, logic, etc. The making of a realistic model of this process is not obtainable today, and we doubt that it will be in the forseeable future.

Within small, well limited domains of knowledge it is, however, possible to build a knowledge base using if-then rules, knowledge bases with surprisingly good reasoning capabilities. An if-then rule has the following general form:

$$\text{premises} \rightarrow \text{conclusion}$$

or, expressed using the syntax of a conventional programming language

```
if
  premise and
  premise …
then
  conclusion
```

We stipulate that the premises have the form

$$\text{premise} \wedge \text{premise} \wedge \ldots \wedge \text{premise} \rightarrow \text{conclusion}$$

and that only one conclusion is made in every rule[2].
A rule of the form

$$\text{premise}_1 \vee \ldots \vee (\text{premise}_{n-1} \wedge \text{premise}_n) \rightarrow \text{conclusion}$$

is then transformed into the rules

[2]A rule with several conclusions then has to be split into several rules, all with the same premises, but with different conclusions.

$$\begin{aligned} &\text{Rule}_1 : \text{premise}_1 \rightarrow \text{conclusion} \\ &\vdots \\ &\text{Rule}_k : (\text{premise}_{n-1} \wedge \text{premise}_n) \rightarrow \text{conclusion} \end{aligned}$$

Rules have two parts
- the left hand side, the premises
- the right hand side, the conclusion

Using rules such as this and a logical rule – or rather meta-rule – used already by the ancient Greeks, Modus Ponens, new knowledge could be inferred from old.

Modus Ponens says that

$$((\text{premises} \rightarrow \text{conclusion}) \wedge \text{premises}) \Rightarrow \text{conclusion}$$

i.e.

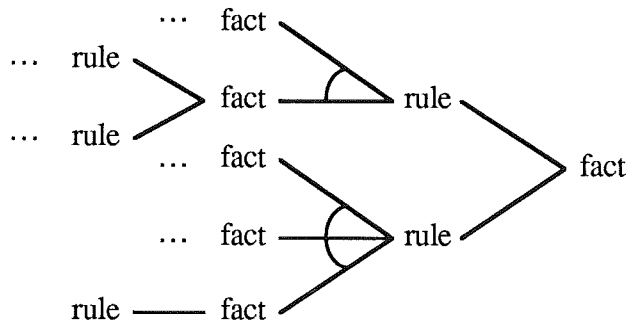
If we have a rule

```
if
  premises
then
  conclusion
```

and we know the premises to be true, then we can infer conclusion as a fact.

This may sound obvious, but it lets us deduct facts mechanically, without interpreting the meaning of neither premises nor conclusions. There is a result from the theory of logic saying that the syntactically derived theorems are the same as the semantically showed tautologies.

We have used the term fact to denote premises as well as conclusions[3]. The conclusion of one rule may be a premise in other rules, and this way the rules are glued together by facts. This connection between rules and facts may be illustrated as below.



The deduction tree is an AND-OR-tree. Since all premises of a rule must be valid the rule-level is an AND-node whereas only one of

[3]Strictly speaking, a conclusion or a premise waiting for its verifiation is, of course, not a fact.

the rules infering a given fact must be valid to make the fact true, thus making the fact-levels OR-nodes.

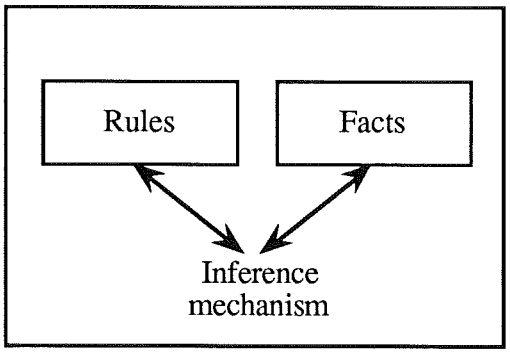
There are two principally different kinds of facts in our system:

- base facts
- derived facts

Base facts are the leaves in the AND-OR-tree, the atoms of inferred knowledge. These must be supplied from outside the system, without the surveillance of the inference mechanism.

Derived facts are facts inferred from rules, i.e. those facts not beeing base facts.

The system, built by base facts, rules, derived facts, and inference mechanism, may be pictured as below:



The reasoning machine as a production system.

In our system we have modelled knowledge in a fashion similar to an AND-OR-tree. Every rule and every fact, base facts as well as derived facts, is an object. They are linked to each other as in the AND-OR-tree. The inference mechanism could be interpreted as the connection between facts and rules and is supplied by the system.

This way of specifying a system is said to be *declarative*, as opposed to procedural, i.e. the user says *what* he wants do be done, not *how* something should be done.

3.2 Different kinds of reasoning

When reasoning one can, as indicated above, using known facts deduct conclusions, thus generating new facts. This kind of reasoning is called *forward chaining*, one infers from premises to conclusions.

Another way of reasoning is to start from a hypothesis, a fact one wants to prove, and try to prove it backwards. This is done by, if the hypothesis is not an already known fact, examining the rules inferring the hypothesis as a conclusion. The premises in those rules are new hypotheses, and is proved in the same way as the original hypothesis. When all premises in a rule inferring the hypothesis has been proven the hypothesis is proved. This way of reasoning is called *backward chaining*.

Since backward chaining plays such an important rôle in our thesis we describe it in a small example.

Assume we have the following rules:

```
Half square rule:
  if
    triangle is isosceles and
    triangle is right
  then
    triangle is half square

Isosceles rule:
  if
    triangle has two equally long sides
  then
    triangle is isosceles

  Right rule:
  if
    triangle has a 90° angle
  then
    triangle is right
```

We also have a triangle with two equally long sides and a 90° angle, i.e.

```
(fact 1)  triangle has two equally long sides
```

and

```
(fact 2)  triangle has a 90° angle.
```

We now ask ourselves if the triangle is a half square?

```
(goal hypothesis )    triangle is half square
```

First we examine if this hypothesis is a known fact, which i turns out not to be.

Then we examine our rules, and see if there is any rule which may conclude our hypothesis, and there is – the half square rule. This rule says that a triangle is a half square if the following hypotheses hold

```
(hypothesis 2)   triangle has two equally long sides
(hypothesis 3)   triangle is right
```

Now, to prove our goal hypothesis we have to prove these new hypotheses.

To prove these hypotheses we do exactly the same thing as we did to prove our goal hypothesis. First we se if any hypothesis is a known fact, and if not all hypotheses are found to be known fact we start looking for rules which may conclude our remainant hypotheses.

In our example it turns out that hypothesis 2 is known as fact 1 and hypothesis 3 is known as fact 2. As this is the case we may eliminate them from the hypothesis stack, and infer that our triangle is a half square since the half square rule is applicable.

The backward chaining may seem awkward in this small sample, but in a realistic system it often proves to be the most efficient way

of reasoning, since uninteresting paths of reasoning may be pruned off[4].

If you recall the comparison of inference with an AND-OR-tree, we found out that backward chaining on a hypothesis may be thought of as walking backwards in the AND-OR-tree, from hypothesis to basefacts.

3.3 Multi object systems

As mentioned already in the introduction, one of our objectives was to develop a system capable of treating any number of objects and compare these objects to each other in the rules.

In the following section we show how we devised such a system. The key to our solution of the problem is to "free" the objects to be examined from the rules.

A rule in a traditional expert system may look like this:

```
If
   Tank_B.Waterlevel > Tank_B.ReferenceValue
then
   Tank_B.flow.stop
   TooFull Tank_B
```

The premise of the rule is a logical condition on an object tied to the rule. If our system has more than one tank we write similar rules for the different tanks.

Having pondered the idea of iterative rules and rules applied to larger data structures we finally came up with a conceptually simple method for treating an arbitrary number of objects. The idea was to let every rule have a mechanism which fetched all objects to be examined. The rule above would then look like this:

```
If
   there is
      CurrentTank(Tank)
   and
      Tank.Waterlevel > Tank.ReferenceValue
then
   action
      Tank.flow.stop
   note fact
      TooFull(Tank)
```

i.e., the objects become variables in the rules. If the system is told which tanks are interesting, this one rule will treat them all.

The left hand side – the premises – in rules of this kind are divided into two parts:

- existence section and

[4]By using variables not initialized until used, one may avoid inputing data not really needed when backward chaining, thus making a substantial gain in system performance.

We have implemented such variables in a toolbox for our system, for further details see appendix–A.

- logical section.

The existence section collects the objects to be examined whereas the logical section contains the logical condition which singles out those of the objects satisfying the rule's condition.

The rules can generate different instances of facts, conclusions, for different sets of objects. The term fact is therefore not unambiguous anymore if we do not specify for what set of objects it is satisfied. A fact which is verified for some set of objects may be false for another set of objects.

With a system such as this one can handle any number of objects, without having to specify the number before runtime.

There are some other attractive features with a system of this kind, as illustrated in the following two examples:

Example

```
if
   there exists
      ObjectKind(obj1)
      AnotherObjectKind(obj2)
   and
      logical condition
   then
      note fact
         Combination(obj1,obj2)
```

This rule gives all possible combinations of objects of the two kinds.

Another example

```
if
   there exists
      ObjectKind(obj)
      AnotherObjectKind(obj)
   and
      logical condition
   then
      note fact
         SomeFact(obj)
```

This rule gives all objects being of both kinds.

3.3.1 Active backward chaining

When infering backwards in a traditional system the user begins with a hypothesis and tries to find a valid proof[5] of it. Since facts and rules in these systems are unambiguous the meaning of this gives rise to no ambiguity.

In our system a fact is determined by a set of objects. The backward chaining should thus be made on an instance of the fact instead of on the fact itself. Backward chaining on a fact with a certain set of objects could look like this:

---

[5]A proof is a chain of rule applications leading to the hypothesis.

```
fact(Obj1, Obj2, ...).bc
```

The result of this backward chaining would be a verifiation or falsifiation of the hypothesis.

The user might, however, be interested in knowing if a fact may be verified for a set of objects where at least one of the objects is arbitarily chosen. To do this she could, of course, generate all possible such sets of objects and try to prove the corresponding facts, but it would be better if she could write

```
fact(..., WildCard, ...).bc
```

where WildCard denotes any object and is put in those positions where any object would do. The result of this operation could be, as earlier, a verifiation or falsifiation, but better still would be if the backward chaining returned the set of WildCard objects satisfying the hypothesis. If the returned set is empty, then the hypothesis is falsified for every possible set of objects, i.e. a non-empty set could be interpreted as a verifiation of the hypothesis in the traditional sense.

We found this latter method so interesting that we decided to make all objects WildCards. This means that the operation backward chaining in our system is not a boolean function but rather a 'set of sets of objects'-function, actively finding sets of objects verifying hypotheses. To perform a backward chaining search the user writes

```
fact.bc
```

and gets a list, possible empty, of sets of objects verifying the fact.

To make a distinction from the traditional backward chaining method we have called our method active backward chaining[6].

3.3.2 The need for actions

When the left hand side of a rule is satisfied for a set of objects an instance of a fact is generated. It may sometimes be of interest to the user to execute some program statements when this occurs. She may want to create an object to propagate in the inference, write something, update some global database or, though God on principal so forbid, interfere with the work of the inference mechanism in some ingenious way.

To enable this we decided to add an action-section to the rules, i.e. a possibility for the user to specify program statements to be performed once a fact is generated.

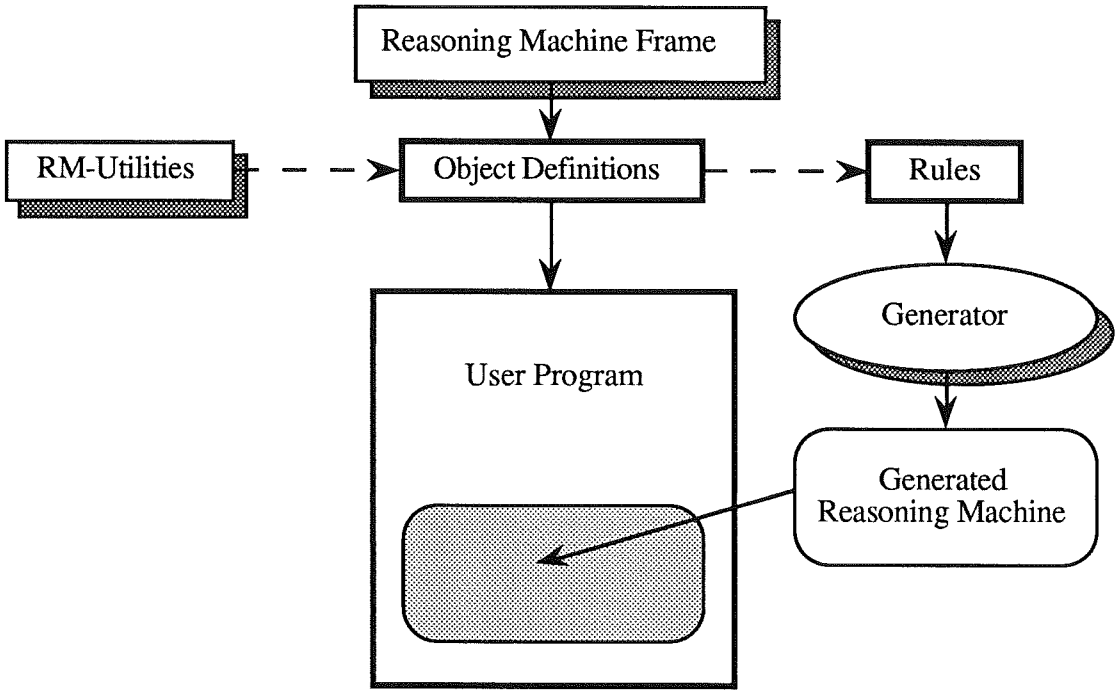
---

[6]The method has many similarities with PROLOG [Clock81], a similarity we did not discover until the method was devised!

Our method for propagating the objects is, however, quite different from that used by common PROLOG-systems.

3.4 Specification of our system

Based on what has been said above, we decided to develop a system with the following basic structure



The picture is to be understood in this way:

The user creates three files:
Object Definitions:

Here the user defines the objects he wants to use in his system. They are defined in a Object Definition Class. This class has the prefix ReasoningMachineFrame, which is provided by our system.

Rules:

The rules and base facts to be used by the reasoner. Their syntax and semantics are described below. This file is compiled into a reasoning machine class by the generator.

User program:

The user's main program wherein an instance of the generated reasoning machine is created. To this instance the user program sends base facts and when all base facts are sent the inference may be activated by a backward chaining on any defined fact.

In fact, if the user wants to, she may use several different reasoning machines in the same user program, thus implementing some kind of expert committee.

3.4.1 Rule definition

The reasoning machine has three kinds of components:

- Base facts
- Derived facts
- Rules

In the rule definition file the user declares only base facts and rules. The derived facts follows from the right hand side of the rules.

Base facts are defined this way:

```
##Basefact BasefactName(parameters);
  #Var
    declaration of parameters
##End
```

The syntax of the rules are

```
##Rule Rulename
  #Var
    ! Declaration of the objects ;
    ! occuring in the rule ;
    ref(...) ...
  #Exists
    ! The existence part of the left hand side ;
    premise1(...)
    premise2(...)
    ...
  #And
    ! Logical condition expressed ;
    ! in the implementation language ;
  #Implies
    #Action
      ! Statements expressed in the ;
      ! implementation language ;
    #Fact
      ! Derived fact ;
      conclusion(...)
##End
```

The syntax is, as shown, very rudimentary. What would have made it difficult to use a syntax more in style with ordinary programming languages is that we in the rules read expressions and statements in the implementation language. If we should parse these expressions and statements the complexity of our parser would have increased considerably, without any gain in the overall performance of the system.

The rule name is an arbitary name, used only to make the generated code easier to read. Every rule name must be unique.

In the #Var-section the objects to be handled in the rule are declared, also objects only used in actions must be declared.

The #Exists-section holds the existence-part of the left hand side of the rules, i.e. it selects the objects to be used in the ...

#And-section. The logical condition of the action-section is expressed in the implementation language, and may be as simple or complicated as the user wants it to be, only the implementation language puts limits on it.

The reserved word #Implies has as its sole purpose to mark the border between the left hand side and the right hand side of the rule.

In the #Action-section, which is optional, the user may insert those program statements she wants to be performed every time a conclusion is drawn.

The #Fact-section describes the generated conclusion and specifies its set of objects.

To use a generated reasoning machine the user writes in her program

```
<ObjectDefinitionsClass>
begin
   ...
   ref(<Reasoning Machine Name>) Reasoner;
   ...

   Reasoner :- new <Reasoning Machine Name>;
   ...
```

To notify the reasoner of what objects it should handle, for every object, the user writes

```
Reasoner.<Base Fact Name>Fact.NoteFact(<object>);
...
```

and then, to perform a backward chaining call on a fact, she writes

```
result :- <fact>Fact.bc;
```

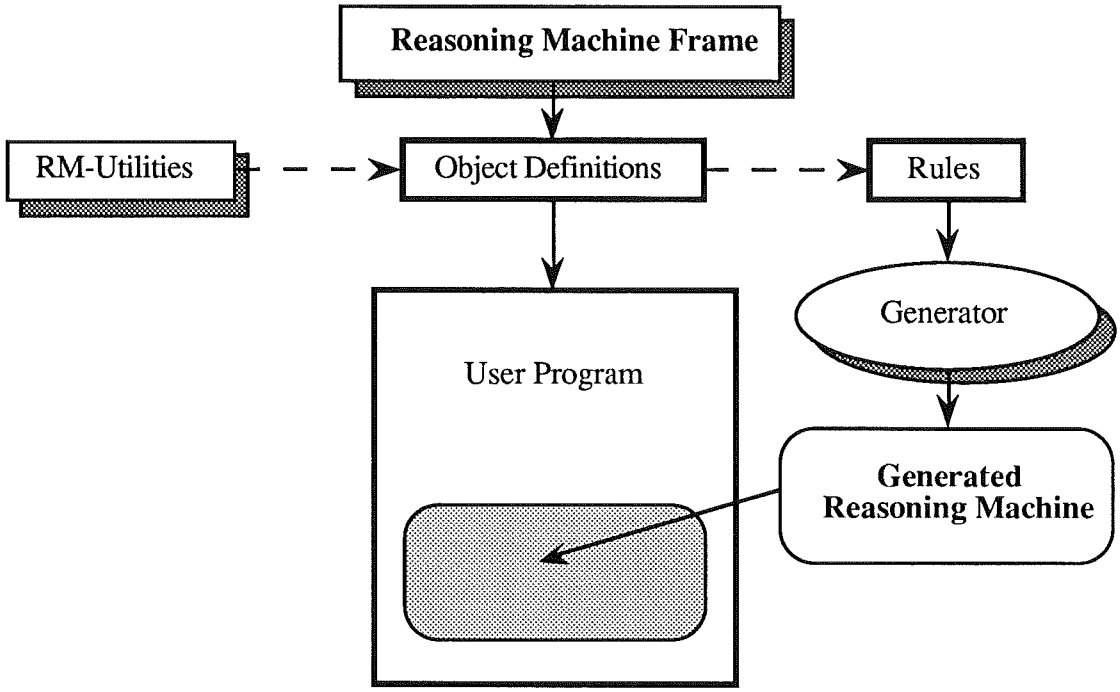
In return she gets a ref(ListOfList) object containing a list of the sets of objects verifying the fact.

When the user defines her rules, she must make sure that there are no circular dependencies amongst her rules. We also made a restriction in the #Exists-section of the rules, no object may occur more than once in every premise. This restriction could have been avoided by a slight change in our backward chaining procedure, but we thought the gain would not compensate for the added complexity of the backward chaining procedure.

4 Implementing our system

In this chapter we will describe the methods we developed for achieving a system such as described in the summary of the previous chapter.

The picture below illustrates the overall structure of the system.



System organization[7]. In this chapter we will take a closer look at the generated reasoning machine.

This scheme has been explained in the summary of the previous chapter. In this chapter we will briefly describe the Reasoning Machine Frame and then explain what methods are used in the generated reasoning machine.

The implementation of the generator is described in the next chapter whereas the Reasoning Machine Utilities are described in an appendix since it not really has to do with the inference mechanism, although they vastly improve the system performance[8].

---

[7]There is a saying that "a picture says more than a thousand words". By using this picture twice we have saved the reader from reading at least two thousand two words.

[8]The Reasoning Machine Frame contains methods to provide the user with a kind of "lazy" instantiation of variable-values, i.e. a variable has not to be given their value until used. By using variable of this kind the user may avoid answering questions not necessary for the reasoning.

4.1 System organization

Our basic idea was to generate a class, a reasoning machine, which could examine objects in the users program. To make the users objects referenceable from inside the reasoning machine they must have some superclass known to the reasoning machine. This superclass, Object, is defined inside ReasoningMachineFrame, together with classes for handling sets of such objects, and sets of such sets. The class ReasoningMachineFrame therefore must be used as superclass on the class where the user defines his objects.

The basic components of the reasoning machine are, as mentioned before, the

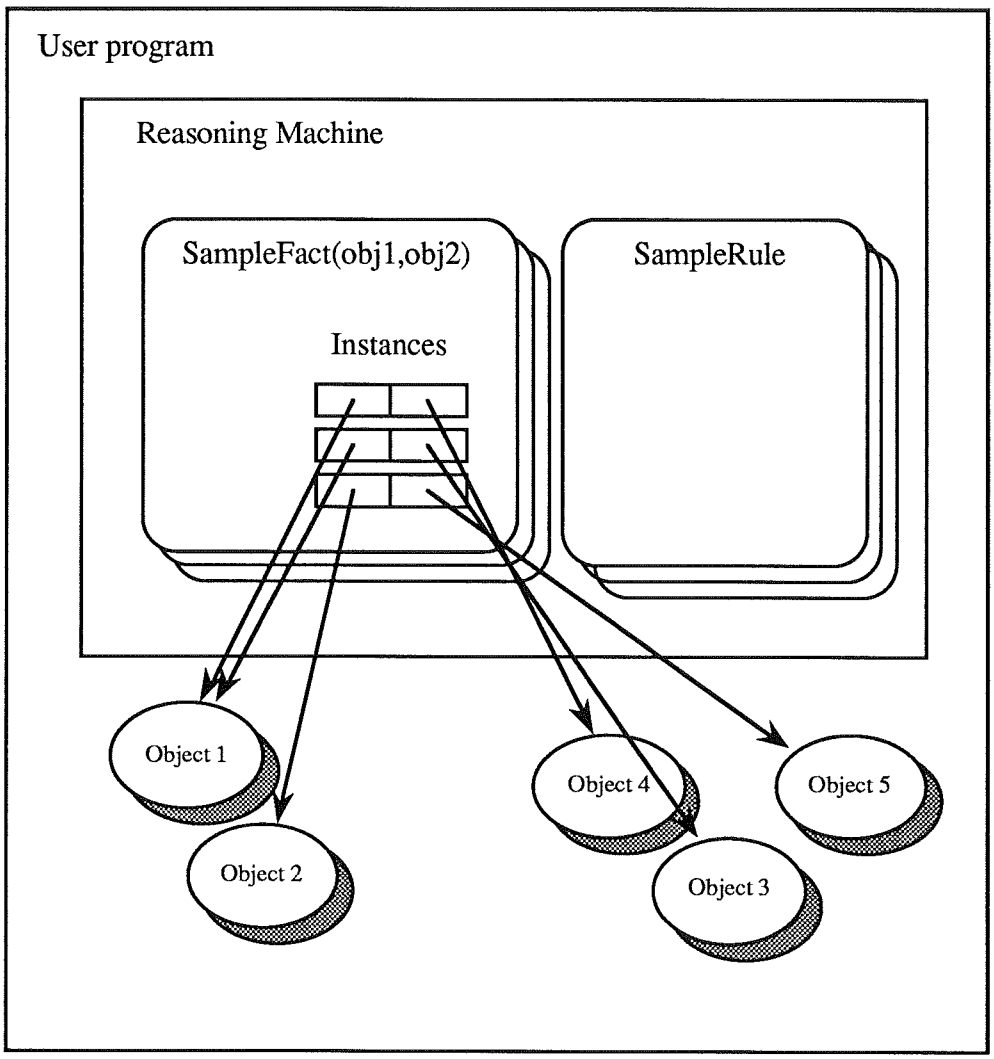
- *base facts,*
- *rules,* and
- *derived facts.*

When we speak of a fact we mean a description of a fact, not one specific instance of one fact.

In our system we describe all basefacts, rules and, derived facts by specific classes, each of whom has one instance. The instances of fact classes have references to all their instances once their backward chaining procedure has been called[9], as illustrated below.

This is, of course, not to be confused with the term "lazy evaluation", although there are similarites.

[9]The first backward chain call on a fact gives rise to backward chaining calls to all rules implying the fact. Once this has been done, the fact remembers all the instances received and returns them directly on the next backward chaining call, thus making the system more efficient.



This means that every fact is described by one object. This object has, in turn, references to all the objects in the user program which makes the fact valid.

The classes have the following methods:

Base facts:

Init
NoteFact(set of objects)
BackwardChain

Rules:

BackwardChain

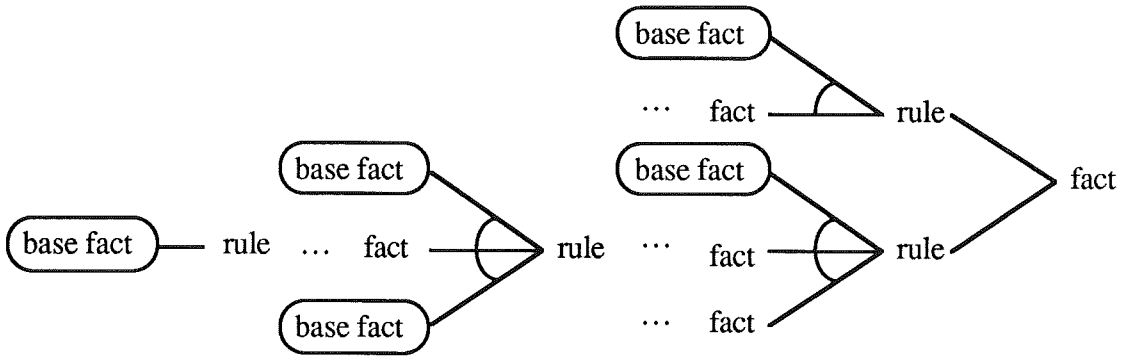
Facts:

Init
BackwardChain

A base fact or derived fact with the name <factname> is described in a class called <factname>Fact__ whereas a rule with the name <rulename> is described in the class <rulename>Rule__. When the system is initialized one instance of every fact and rule, with the name <factname>Fact or <rulename>Rule, is created.

The objects describing base facts, rules, and derived facts are connected as in an AND-OR-tree described in the prevoius chapter,

i.e. the inference mechanism is hard-coded in the bc-methods of the objects.



A traditional expert system does not have its inference-links computed before runtime. Some readers might say that our approach is not as general as the inference mechanism of a common expert system, but we think this is only a question of perspective.

4.2 Methods for the active backward chaining

In the following section we will describe the methods we devised for the backward chaining and how to implement them. Our method is based on the view of the inference mechanism as an AND-OR-tree. A fact is true if *any* of the rules infering the fact is applicable. In order for a rule to be applicable *all* its premisses, in some meaning, must be valid.

Backward chaining on a derived fact should return all sets of objects verifying the fact. The result of backward chaining on a fact is thus the union of sets of objects making any of the rules infering the fact applicable. Once a fact has been backward chained its instances are stored – in order to speed up the execution.

Even simpler is the backward chaining on a basefact, which just returns all instances of the basefact imported from outside the reasoning machine.

Alas, the backward chaining on a rule is somewhat more complicated, and we devote the remainder of this section to it.

As mentioned above, the rules of our system have the following general form:

```
##Rule RuleName
  #Var
    ...
  #Exists
    ...
  #And
    ...
  #Implies
    #Action
```

```
        ...
      #End
      #Fact
##End RuleName
```

The left hand side has two parts, the #exists-section and the #and-section. The #exists-section may be regarded as a mechanism for collecting those objects to be tested by the logical expression in the #and-section. It soon became evident that the backward chaining method of the rules should have the following general form

1. Collect all sets of objects obeying the requirements of all premisses in the #exists-section
2. For each of these sets, apply the logical term of the rule. Eliminate those sets which do not satisfy the logical term.
3. Using a template of the fact to be inferred, select the appropriate objects from the sets which have been let through in point 2, and return them as the result of the backward chaining.

Points 2 and 3 are quite straightforward to implement, but point 1 deserves some more attention. We therefore now turn our attention to the #exists-section of the rules.

4.2.1 Difficulties when collecting objects in the existence-section of a rule

To indicate the difficulties put forward to one when collecting the objects in the existence-section we have made some small examples.

Example 1

```
f1(p1)
f2(p1)
```

Example 2

```
f1(p1)
f2(p2)
```

Example 3

```
f1(p1,p2)
f2(p1,p3)
f3(p2,p3)
```

In the first example the following holds: Facts f1 and f2 are satisfied by one set of objects each. The rule demands that objects (p1) must satisfy both f1 and f2. Thus only objects contained in both sets – the intersection of the sets – are let through.

In the second example we also have two different facts and thus two different sets of objects satisfying facts. This time, however, the rule does not demand that objects (p1 and p2) must satisfy both facts. Instead we must try the logical terms of the rule with all combinations of two objects such that the first element is a member

of the first list and the second object is a member of the second list, i.e. the cartesian product of the two lists.

The third example is considarably more difficult than the first two. The objects to be tested by the rules logical term are three in number, and the triples must obey the following condition: Its first element must satisfy f1 together with an object p2 such that p2 satisfies f3 with an object p3 which also satisfies f2 together with p1.

Steps 1 to 4 in the next section describes one of our solutions to this problem[10].

4.2.2 Detailed description of the backward chaining method

The result of the backward chaining on a fact is the set of sets of objects satisfying the fact. We decided to implement these sets as lists, i.e. we made one structure called List which could contain a set of objects and one structure called ListOfLists able to contain a set of sets of objects. These classes have been defined in the ReasoningMachineFrame. A more detailed description of these classes can be found in the next chapter, in this chapter we will assume that convenient operations have been defined for the classes.

At an early stage we decided that the backward chaining should be based on the transformation of these list structures.

To simplify things – remember the Boltzmanian words in the preface – we describe the backward chaining method by an example.

```
##Rule SampleRule
   #Var
      ref(Class1) Obj1;
      ref(Class2) Obj2, Obj3, Obj4;
   #Exists
      fact1(Obj1,Obj2);
      fact2(Obj2,Obj3);
      fact3(Obj1);
   #And
      SimulaCondition(Obj1,Obj2,Obj3)
   #Implies
      #Action
         Obj4 :- new Class2;
      #Fact
         SampleFact(Obj1,Obj4);
##End RuleName
```

We commence the description by making some general comments on the rule:

- It has four objects, three of which are of the same class.
- All objects except Obj4 are used in the #exists-section.
- The object Obj4 is created in the #Action-section. An object declared in the rule, but not used in the #exists-section must be set either in the #And-section - which is highly unusual - or in

[10]We made another, conceptually simpler, solution as well but decided not to use it on grounds of inefficiency.

the #Action-section, otherwise its declaration would have been unnecessary.

- The generated fact involved only Obj1 and Obj4.
- The #exists-section has references to facts fact1, fact2 and fact3, i.e. these facts are the premises of the rule.
- The #and-section has a logical condition expressed in an ordinary programming language.
- It has an #action-section consisting of ordinary programming statements.

Backward chaining on this rule should give us a set of tuples of objects, such that the objects simultaneously fulfils the requirements of Obj1 and Obj4 in the rule, #existence-section as well as #and-section.

An object Obj4 is created every time there is a triple of objects Obj1, Obj2 and Obj3 passing through both parts of the left hand side of the rule, it therefore is not used in either of the two parts of the left hand side of the rule.

Our basic idea was to create a rule class with a method bc with the following general form:

a. declaration of objects used in the rule
b. backward chaining calls to premises of the rule, yielding sets of sets of objects
c. operations to transform these sets into one set of sets of objects passing through the entire #exists-section of the rule
d. for all sets in this set do
   assign rule objects using the present set
   if the rule's logical condition applies for these objects then
      perform actions
      add objects declared in declaration but
      not used in #exists-section to present set
   else
      delete present set from set of sets
e. select those objects to be passed on from sets sliping through step d
f. return those sets

The inference (step b-e above) may be implemented in the following way:

1. Create a template for all objects occuring in existence-section of the rule

$$[obj_{1\_}, obj_{2\_}, obj_{3\_}]$$

2. Perform backward chaining calls on all premisses. Let us assume we get the following result:

$$\begin{array}{ccc} \text{f1TempList} & \text{f2TempList} & \text{f3TempList} \\ \downarrow & \downarrow & \downarrow \\ \left\{\begin{array}{l}[obj_{11},obj_{21}]\\ [obj_{11},obj_{22}]\end{array}\right\} & \left\{\begin{array}{l}[obj_{21},obj_{31}]\\ [obj_{22},obj_{32}]\end{array}\right\} & \left\{\begin{array}{l}[obj_{11}]\\ [obj_{12}]\end{array}\right\} \end{array}$$

3. Now rearrange the backward chaining lists to fit the template above. In our example f1TempList does not have any obj3-elements, f2TempList does not have obj1-element and so forth. These missing objects are replaced by obj__-elements which will serve as a kind of vacancies. When all premiselists have been rearranged we have

$$\begin{array}{ccc} \text{f1List} & \text{f2List} & \text{f3List} \\ \downarrow & \downarrow & \downarrow \\ \left\{\begin{array}{l}[obj_{11},obj_{21},obj\_\_]\\ [obj_{11},obj_{22},obj\_\_]\end{array}\right\} & \left\{\begin{array}{l}[obj\_\_,obj_{21},obj_{31}]\\ [obj\_\_,obj_{22},obj_{32}]\end{array}\right\} & \left\{\begin{array}{l}[obj_{11},obj\_\_,obj\_\_]\\ [obj_{12},obj\_\_,obj\_\_]\end{array}\right\} \end{array}$$

4. To join these lists into one list - called CheckList, containing all sets of objects satisfying all premises, we now begin by uniting the two first premise lists (if there is only one premise we are already finished). We begin by creating all tuples such that its first element is in the first list and its second element is in the second list, i.e. the cartesian product of the two lists[11]:

$$\begin{array}{c} \left\{\begin{array}{l}[obj_{11},obj_{21},obj\_\_]\\ [obj_{11},obj_{22},obj\_\_]\end{array}\right\} \left\{\begin{array}{l}[obj\_\_,obj_{21},obj_{31}]\\ [obj\_\_,obj_{22},obj_{32}]\end{array}\right\} \\ \Downarrow \\ \begin{array}{l}([obj_{11},obj_{21},obj\_\_],[obj\_\_,obj_{21},obj_{31}])\\ ([obj_{11},obj_{21},obj\_\_],[obj\_\_,obj_{22},obj_{32}])\\ ([obj_{11},obj_{22},obj\_\_],[obj\_\_,obj_{21},obj_{31}])\\ ([obj_{11},obj_{22},obj\_\_],[obj\_\_,obj_{22},obj_{32}])\end{array} \end{array}$$

We then try to unite the lists in these tuples. This is done by pairwise comparison, and is best illustrated by an example.

$$\begin{array}{c} [obj_{11},obj_{21},obj\_] \\ [obj\_\_,obj_{21},obj_{31}] \\ \downarrow \\ [obj_{11},obj_{21},obj_{31}] \end{array}$$

The two lists is compared to each other, element by element. A vacancy (i.e a obj__-element) matches any object, whereas other objects only matches if they are the same. If there is any contradiction, i.e. two elements in the same position being instantiated and different, a null list is returned.

---

[11]Observe that the elements of the lists are lists themselves.

$$[obj_{11}, obj_{21}, obj\_]$$
$$[obj\_, obj_{22}, obj_{32}]$$
$$\downarrow$$
$$[obj_{11}, mismatch, obj_{31}]$$
$$\downarrow$$
$$[\,]$$

A list of all lists but the null lists are returned by the join function.

When the first two lists have been joined together to form CheckList, the third list can be joined to CheckList and so forth. Eventually CheckList contains the sets of sets of objects sought for.

In our sample CheckList will be:

$$\text{CheckList} = \left[(obj_{11}, obj_{21}, obj_{31}) (obj_{11}, obj_{22}, obj_{32}) \right]$$

These are the sets of triples[12] satisfying all terms of the #exists-section.

5. Try the logical term in the #and-section for every set of objects in CheckList.

   For every set obeying the logical condition the following actions are taken:

   a any statements in the #action-section is performed, with the actual values of the objects in the rule set to the values in the present set of objects
   b the objects declared in the #var section, but not used in the #exists-section is appended to the set of objects in CheckList

   The sets of objects not satisfying the logical condition are eliminated from CheckList.

   Let us, in our example, assume that the first triple satisfies the condition, and that the other does not. CheckList will transform into:

$$\text{CheckList} = \left[(obj_{11}, obj_{21}, obj_{31}, obj_{41}) \right]$$

6. CheckList now contains all sets of four objects meeting the requirements of the left hand side of the rule. From each of the sets we pick the objects contained in the generated fact. They are put into OutList

$$\text{CheckList} = \left[(obj_{11}, obj_{41}) \right]$$

[12]The triples are represented as lists.

and it is the result of the backward chaining on the rule.

Note that this method does not require the generator to know anything about neither the logical condition nor the action statements.

5 Generating a reasoning machine

The following chapter shows how to generate the implementation described in the previous chapter. We begin by describing the classes on which the inference engine is built. The remainder of the chapter describes how the code in the generated reasoning machine class is generated.

As mentioned before we chose to implement the system in Simula, but it could have been implemented in C++, Smalltalk, or any other object-oriented language.

5.1 Fundamental classes

The result of active backward chaining is a set of sets of objects or, using a common computational concept, a list of lists of objects. The reason for us to use lists instead of sets is that the ordering of the objects is essential when we want to pick out the objects from a list/set. To implement this we devised three classes, *Object*, *List*, and *ListOfLists*. These classes are defined within the class ReasoningMachineFrame. They will be described briefly here.

```
class Object;
begin
  text ObjectName;
end;
```

The class Object is used as a prefix on the classes the user wants to examine in the reasoning machine. Its purpose is to give those classes a common superclass, known to the List and ListOfLists classes.

```
class List;
begin
  integer procedure Cardinal;
  !-   Gives the length of the list;

  ref(List) procedure emptyList;
  !-   Returns an empty list;

  ref(List) procedure NoneList(length);
    integer length;
  !-   Returns a list of length length
    with all elements nones;

  procedure Append(anObject);
    ref(Object) anObject;
  !-   Appends anObject to the end
    of the List;

  ref(Object) procedure ObjectNo(no);
    integer no;
  !-   Returns object number no
```

```
      in then List;

    ref(List) procedure Collide(otherList);
      ref(List) otherList;
    !-   Performs the operation Collide
      with the List and otherList;

    ref(List) procedure Copy;
    !-   Gives a copy of the List;

    procedure Print;
    !-   Makes a rudimentary print
      of the ObjectNames of the
      objects in the List;
end;
```

A set of objects, of arbitary cardinality, is represented using the class List. The different instances of a fact are references to Lists.

```
class ListOfLists;
begin
  integer procedure Cardinal;

  ref(List) procedure ListNo(no);
    integer no;

  ref(ListOfLists) procedure emptyListOfLists;

  ref(ListOfLists) noneList(length);
    integer length;

  procedure Append(aList);
    ref(List) aList;

  procedure AddNones;

  procedure Add(aListOfLists);
    ref(ListOfLists) aListOfLists;

  ref(ListOfLists) procedure Select(no);
    integer no;

  procedure Eliminate(no);
    integer no;

  ref(ListOfLists) procedure Join(otherListOfLists);

  ref(ListOfLists) procedure Union(otherListOfLists);

  ref(ListOfLists) procedure Copy;

  procedure Print;
end;
```

The class ListOfLists is used to represent the result of a backward chaining. We have put the method Join into this class, to simplify the bc-methods of the rules. Some of the other methods are also put here for simplificational reasons. AddNones and Add, eg, are short-hand for constructs which would have made the bc-methods of the rules more difficult to read.

5.2 Generated code

The generated code can be divided into five parts

- Reasoning Machine heading, etc
- Rule classes
- Basefact classes
- Fact classes
- Initialization of facts and rules

The generated class has the following general form – note that the superclasses Fact, BaseFact and Rule have no actual meaning:

```
external class ObjectDefinitions;
ObjectDefinitions class ReasonerName;
begin
   class Fact;;
   class BaseFact;;
   class Rule;;

   Rule class SampleRule__;
   begin
      ref(ListOfList) procedure bc;
      begin
         ...
      end bc;
   end SampleRule__;

   ... all other rules ...

   BaseFact class SampleBaseFact__;
   begin
      ref(ListOfLists) myInstances;

      procedure NewFact(...);
         ref(...)...;
      begin
         ...
      end;

      ref(ListOfLists) procedure bc;
      begin
         ...
      end bc;
   end SampleBaseFact__;

   ... all other base facts ...

   Fact class SampleFact__;
   begin
      boolean activated;
      ref(ListOfLists) myInstances;

      ref(ListOfLists) procedure bc;
      begin
         ...
      end bc;

      ...
   end SampleFact__;
```

```
    … all other derived facts …

    ref(SampleRule__) SampleRule;
    …

    procedure InitRules;
    begin
      SampleRule :- new SampleRule__;
      …
    end InitRules;


    ref(SampleBaseFact__) SampleBaseFact;
    …

    procedure InitBaseFacts;
    begin
      SampleBaseFact :- new SampleBaseFact__;
      …
    end InitBaseFacts;


    ref(SampleFact__) SampleFact;
    …

    procedure InitFacts;
    begin
      SampleFact :- new SampleFact__;
      …
    end InitFacts;

    procedure Init;
    begin
      InitRules;
      InitBaseFacts;
      InitFacts;
    end;

end ReasonerName;
```

This template is then filled in with code specific to the generated reasoner. In the next few sections we will show how this code is generated.

5.2.1 Reasoner Heading

In the head of the rule definition file the name of the reasoner and the name of the object definition file are declared. These names are just pasted in in the appropriate places in the heading of the generated reasoning machine class.

The classes BaseFact, Rule, and Fact are then declared. They are used for two reasons

- they clarify which classes describes basefacts, rules, and derived facts
- sentimental reasons

We have to admit that the latter reason is the stronger.

5.2.2 Rule Code

The rule classes contains, as mentioned previously, only the method for backward chaining. It is generated as follows:

- Generate rule heading

```
Rule class <RuleName>Rule__;
begin
```

- Declare method heading

```
ref(ListOfLists) procedure bc;
begin
```

- Declare all objects declared i the #var-section of the rule.
- For each premise in the #exists-section: declare

```
ref(ListOfLists)
  <premise>List,
  <premise>TempList;
```

- Declare

```
boolean continue;
ref(ListOfLists)
  CheckList,
  BCList;
integer i;
```

- Generate

```
continue := TRUE;
```

  and a backward chaining call on the *first* premise of the rule:

```
<premise>TempList :- <premise>Fact.bc;
continue :=
  continue and
  <premise>TempList.Cardinal > 0;
```

  Continue is used to stop the inference if there is a premise with no valid instances. It is used to prevent backward chaining on premises in vain, but it does not alter the result of the backward chaining.
- For the rest of the premises, generate

```
if continue then
begin
  <premise>TempList :- <premise>Fact.bc;
  continue := premissTempList.cardinal > 0;
end;
```

- It is now time to generate code for the rearrangement of the lists. Generate

```
if continue then
```

```
begin
```

and then, for every premise, reform <premise>TempList according to step 3 in the description of the backward chaining method, i.e.

1. First instantiate <premise>List:

```
<premise>List :- <premise>TempList.Select(<no>);
```

   where <no> is the position of the first element in the template in the backward chaining list for <premise>. If the first element in the template is not used in <premise>, generate instead

```
<premise>List :- new
ListOfLists.NoneList(<premise>TempList.Cardinal);
```

   which is a list of nones, equally long as the number of sets found when backward chaining on <premise>.
2. Then, for each of the following elements i the template generate

```
<premise>List.Add(<premise>TempList.Select(<no>);
```

   if the element exists in <premise>, with <no> declared as above,

```
<premise>List.AddNones
```

   otherwise.

- when these lists now have found their values it is time to join them together to form a list of all lists of objects satisfying the #exists-section of the left hand side of the rule. This list is kept in CheckList, so first generate

```
CheckList :- <premise>List;
```

  where <premise> is the first of the premises. Then, for each premise generate

```
CheckList :- CheckList.Join(<premise>List);
```

- now we will generate code to try the rules #And-section.

```
for i := CheckList.Cardinal step -1 until 1 do
begin
```

  Generate assignment of the objects, using the template describing the order of the objects in the lists. For each variable in the template generate

```
<variablename> :- CheckList.ListNo(i).ObjectNo(<no>);
```

  where <variablename> is declared as object number <no> in the template.

Paste in the rule's logical condition, without interpreting it

```
if (<logical condition>) then
begin
```

and paste in any actions of the rule

```
<action statements>
```

If there are any objects not used in the #exists-section these are now appended to the present list in CheckList, so for all such objects do

```
CheckList.ListNo(i).Append(<object>);
```

if the rules logical condition was not met for the present list it should be eliminated. this is done by

```
    end
    else
      CheckList.Eliminate(i)
end for;
```

- we now generate code for selecting those objects to be passed on. By comparing the template used above, enlargened with those elements appended in the then-part of the logical condition, with the object list of the generated fact we generate the following code

```
BCList :- CheckList.Select(<no>);
```

where <no> is the first of the output objects' position in the template. For each of the other output objects generate

```
BCList.Add(CheckList.Select(<no>));
```

- to return the appropriate list of lists we now simply generate

```
bc :- BCList;
```

and put an end to the if continue statement used to prevent unnecessary pruning by generating

```
end
else
  bc :- new ListOfList.emptyListOfLists;
```

- This completes the backward chaining procedure, and the entire rule class, so we generate

```
  end bc;
end <Rulename>Rule__;
```

5.2.3 Base fact code

The classes describing base facts contains

- the attribute myInstances – holding the instances of the base fact,

- procedure NewFact – putting new instances of the base fact into the attribute myInstances,
- the ref(ListOfLists) procedure bc - which just returns a copy of myInstances, and
- a class body where myInstances is instantiated to an empty ListOfLists.

Only procedure NewFact is worthwhile describing:

1 Generate the heading:

```
procedure NewFact(
```

and write all objects in the declaration of the base fact, followed by

```
);
```

2 Declare all objects in the base fact.

3 Generate

```
begin
  ref(List) myList;

  myList :- new List.emptyList;
```

4 For all objects in the base fact generate

```
  myList.append(<objectname>);
```

5 Generate

```
  myInstances.append(myList);
end;
```

5.2.4 Fact Code

The fact classes are about as simple as are the base fact classes. They contain

- the boolean attribute activated – telling whether the facts has been backward chained yet.
- the ref(ListOfLists) attribute myInstance – containing the sets of sets of objects returned the last time the fact was backward chained. Before the first backward chaining call on the fact this ListOfLists is empty.
- a ref(ListOfLists) procedure bc – performing a backward chaining call if the attribute activated is false, returning a copy of myInstances if activated is true.
- a class body, where activated is set to false.

The following code is generated:

- Class heading

```
Fact class <factname>Fact__;
begin
  boolean activated;
```

```
    ref(ListOfLIsts) myInstance;

    ref(ListOfLists) procedure bc;
    begin
      ref(ListOfLists) outlist;

      if not activated then
      begin
```

Assume rule <rule> is the first of the rules in the rule definition file generating the fact in question:

```
        outlist :- <rule>Rule.bc;
```

Then for every other rule generating the fact:

```
        outlist :- outlist.union(<rule>Rule.bc);
```

The rest of the code is

```
        myInstances :- outlist.copy;
        activated := TRUE;
      end
      else
        outlist :- myInstances.copy;

      bc :- outlist;
    end;

    activated := FALSE;
  end <Fact>__;
```

... and that's all folks, no more fact code has to be generated.

6 Enhancements

Since our system was made as small as possible, without interfering with our main objectives, numerous enhancements suggest themselves.

Many expert systems enable the user to ask *why* a question is asked, and *how* a conclusion is drawn. We have not implemented these features, since they require the expert system to have control over input from the user. The features could, however, have been implemented by making the ReasoningMachineUtilities compulsory. In order to make the why- and how questions work there must be some shortcut between the reasoning machine and the reasoning machine frame – now they are separated by the user program.

Expert systems are divided into categorical systems and probabilistical systems. In a categorical system all knowledge is 100% certain, whereas knowledge in a probabilistical system may be more or less certain. Our system generates categorical systems. Several different approaches to probabilistical systems have been put forward, one of the most famous systems, Prospector [Duda79] [Hart78], uses necessity- and sufficiencyfactors – factors saying how necessary a premise is and how sufficient a premise is. Changing our system into a system using N-factors and S-factors would not be very difficult. To do it the syntax and semantics of the rules have to be altered and some of the methods for the backward chaining have to be refined, but the overall structure of the system needs no alteration.

We did not allow the object lists in the premises of the rules to have the same object in two positions, as in

```
premise(obj1,…,obj1,…)
```

It is quite straightforward to implement this feature. We do not, however, think it to be very useful in a practical system.

When the user enters his rules, and makes his object definitions he has none of the exellent graphical utilities provided by Nexpert at his disposal. The development of such utilities would, of course, enhance the user-friendliness of our system vastly.

7 Conclusions

We have created an expert-system like system, and all our original objectives have been accieved.

This means that the user may

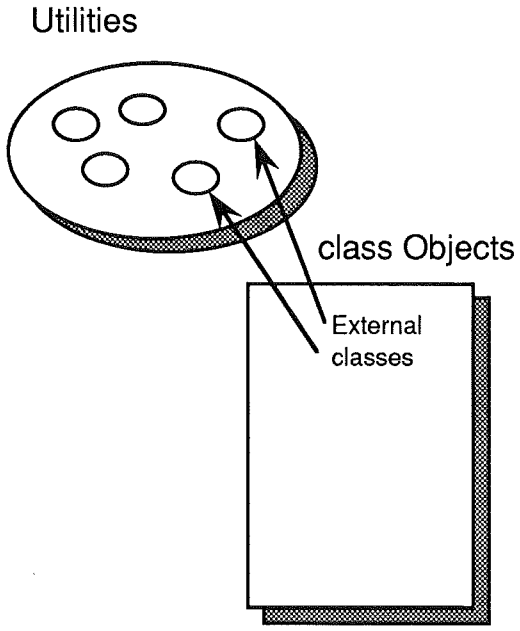
- define her objects in an object oriented programming language
- examine arbitrarily many objects in the system
- combine different objects of the same kind in the rules
- use as complex expressions as provided by the used object oriented programming language
- get a stand alone application which does not require any run-time systems to back it up
- combine several expert systems in one program

Our methods are general enough to be implemented in object-oriented languages such as eg Smalltalk, or C++. The only requirement of the language is that it enables simple inheritance.

We conclude that it is indeed possible to create simple expert system-like systems without too much effort.

Appendix A – Utilities

The Simula language gives a great opportunity to define classes which can be reused later on and facilitate the developement for the knowledge engineer. In our system we have written some classes which can be useful when defining the objects of the system and collected them so that they can be used by future developers. The utilities class library could be extended to a vast number of classes but that we have considered being a work beyond this thesis.



We start below to describe the standard attributes classes in general and continue by giving a more detailed description of the Enumerated Attribute class since this is more complex than the rest. After that we give a short description of how the procedure Boolean Question can be used in a rule.

A–1 Standard Attributes

Using standard Simula types as attributes for your objects is often too primitive. Especially when the values of the attributes are undefined when starting a consultation you feel the need of a mechanism which can fetch the values when they are needed by the system from a certain source. This source can be standard input/output, a file, a database, etc. This behaviour of the different types of attributes can be abstracted to a common superclass which in our system is called Attribute.

A–1.1 Input/Output

The Attribute class interact with its source of information through two channels, one for input and one for output. These two

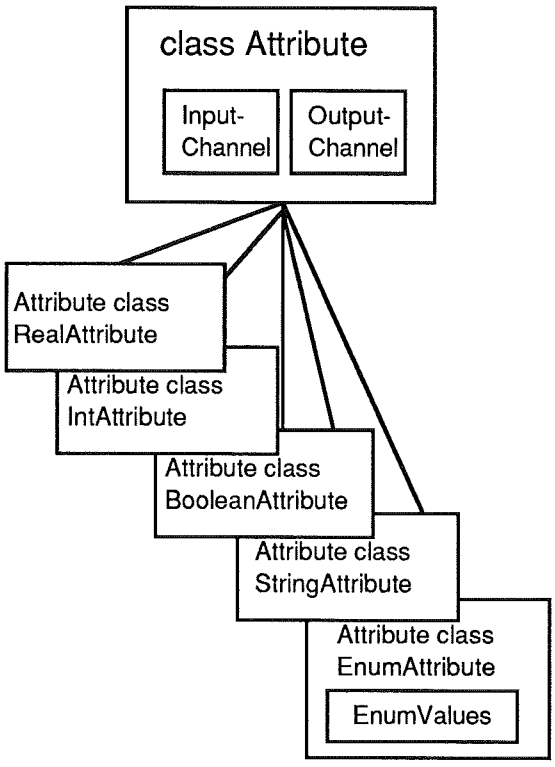
channels are set by the expertsystem builder when constructing the system. One attribute might fetch its value from a file while another one interacts with the user through the screen. The most common way to fetch the values of the values, however, is to ask the user. (Therefore the default values for the channels are standard output and standard input.)

A–1.2 Class hierarchy

The Attribute class has six subclasses, namely:

- class IntegerAttribute
- class RealAttribute
- class BoolAttribute
- class CharacterAttribute
- class TextAttribute
- class EnumAttribute

The five first classes are simply the Simula standard types, enhanced whith the functionality of the attribute class. The last one, EnumeratedAttribute, is a little bit different from the others and is further described below.



The class structure of the standard attributes

All classes have the following operations:

Get – Gets the attribute value. If the value is uninitiated it tries to fetch it from the InputChannel.

SetPromt(newPrompt) – Sets the asktext written on the OutputChannel to newPrompt.

Set(newVal) – Sets the attribute to newVal.

UnSet – Makes the attribute uninitiated.

A–1.3 Enumerated Attributes

The class EnumeratedAttribute class is a little bit more complex than the other ones. An enumerated attribute is really a string whose value can be one of the members in a limited set of strings. This set is represented by an object of the class EnumeratedAttributeValues which is tied to the enumerated attribute in the following way:

```
ref (EnumeratedAttributeValues)  colour;
(...)
ref(EnumeratedAttribute)  CarColour;
CarColour :- new EnumeratedAttribute(colour);
```

The EnumeratedAttribute class has all the standard operations of the attribute classes, extended with the boolean operation equals(intext), which gives true if the value of the attribute equals intext.

The EnumeratedAttributeValues class has the following operations:

- AddValue(newValue) – Adds newValue to the Valuelist.
- Exists(CheckValue) – Checks if CheckValue is in the Set of Values.

A small facility is added to the enumerated attribute class in order to make the system more user-friendly: When the system tries to get the value of an attribute from a user through the screen and the user doesn't answer one of the values in the value-list the whole list is displayed and the user is asked to choose one of the values in the list.

A–2 Boolean Question

Another small utility which is added to the system is the boolean procedure BooleanQuestion. This is very helpful when you have rules which have yes-and-no questions in their terms . As for instance:

```
##Rule MorbusSchlatter
   #var
   ref(Person) P;
   ref(Diagnosis) D;

   #exists
   SubPatella(P)
   #and
   (P.age.get < 22 ) and then
   BooleanQuestion("You feel pain in the "&
                   "knob below the knee cap.") and then
                  ( P.sport.equals("Running") or else
     BooleanQuestion("You feel pain when you start
running.") or else
```

```
    BooleanQuestion("You feel pain when you stress
quadriceps.") )
  #implies
    #action
    D:- new Diagnosis("Morbus Schlatter");
    #end
  #fact
  Diagnosis(D);
##end
```

When this rule is executed by the inference engine the statements inside the parantheses will be displayed after the question: "Is it true that: ". If the user answers "yes" (or anything which begins with a "y" or "Y") the procedure returns true, if he/she answers anything else the procedure returns false.

Appendix B – Applications

In order to test our system we made two applications, an option expert and a sport doctor. The first application mainly tests the multi object aspects and the calculational abilities of the system, whereas the latter shows that also traditional systems can be generated.

B–1 Option expert

For those readers who are not familiar with calls and puts and other things about options, we have made the following overview so that they will better understand the domain of our option expert. The topic is very vast but in order to keep this section reasonably small we have not made the overwiew complete so if any reader would like to start to speculate on the option market we recomend that he or she first studies the information given in [Cox85].

B–1.1 Introduction – What is an option?

An option is the right to buy or sell a certain commodity for a certain price at a certain date. The price is called *striking price* and the day of exercise is called *striking date.* There are to different kinds of options: *calls* and *puts.* The call gives the holder the right to buy *the asset* for the striking price on the striking date.The put gives the holder the right to sell *the asset* for the striking price on the striking date.

Example 1:
A call in a car. The holder has the right to *on* the 21 aug 1990, but not before, buy a Volvo 740 GL for 140 000 SEK. The asset is here a Volvo 740 GL, striking date is 21 aug 1990 and striking price is 120 000 kr. When the exercise is only to be made on the striking date the call is called an *European option.*

Example 2:
A put in a picture. The holder has the right to sell a certain Picasso picture for 15 000 000 SEK before the 1 july 1990. Exercise of the option is allowed at any time. When you can exercise the put at any time it is called an *American Option.*

Options has been used since the 17th century when dutch tulip merchants started to use them. They have also ocurred at an early stage together with cattle dealing in Chicago.

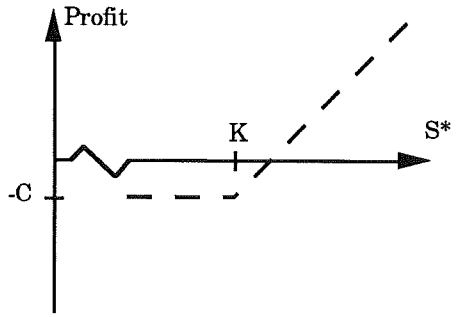
An option can be used in the four following ways:

- It can be exercised
- It can be settled. That is, writer and holder arrange that the holder gets the difference between the value of the asset and the option's striking price.
- It can be sold
- It can expire

B–1.2 Stock options, Index options

A stock option is an option with a stock as the asset. In Sweden there is today, December 1990, one market for dealing with stock options. It is a company which is called Stockholms Optionsmarknad (OM), which provides a market for option dealers. To get a higher volume in the option trading one only uses so called standardized options, i.e. options with certain standardized striking prices and striking dates and only for some certain stocks.

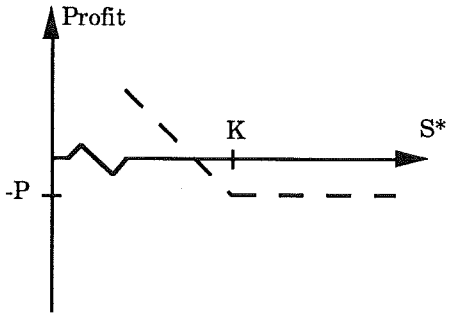
In order to illustrate how an options value on the striking date varies with the asset on the same day you usually draw the following figure :



Profit for call

This figure shows the diagram of a Call. The broken line shows the profit/loss you make on the option as a function of the value of the asset on the striking date. K represents the striking price and C the price you have payed for the call. As the option only is a right to, in this case, buy the share – not a obligation – you never risk to loose more than the C you have payed for the option. The profit can on the other hand, theoretically, be infinitely big.

The corresponding picture for a put looks like this:



Profit for put

P represents here the price of the put. If the price ends over the striking price, you don't exercise the option of course and have only lost P.

An index option is an option where asset is not a stock but an mean value between a number of shares, a so called index. In Sweden, OM provides options for such an index.

B–1.3 Real value and time value

The value of an option in fact consists of two components, the real value and the time value. The real value of a call is determined by the difference between the current price of the stock and the striking price of the option. For the put it is the same operation but with opposite signs.

The longer time it is to the striking date for an option, the greater is the chance the price of the stock will change so the option will be valuable. If we suppose the chance is as great that the price will go in the "right" direction as it is for the "wrong" direction, and as we know our loss in the latter case will never be greater than what we have payed for the option but in the former case it might be infinitely large, the longer the time is for expiration the greater value the option should have. This value is called the time value. In the figures above only the real value is marked as they only show the options's value on the striking date – the time value is then zero.

B–1.4 Black & Scholes' method

There are a number of different theoretical methods for valuating options. The most used is without any doubt the model created by Fisher Black and Myron Scholes 1973, the so called Black & Scholes model.

The factors which affect an options value are, according to Black & Scholes are:

- the value of the asset, S
- the option's striking price, K
- the time to the striking date (usually measured in years), t
- current interest value, r
- the volatily of the asset, σ[13]

The value of a call is, according to Black & Scholes is

$$C = S \cdot N(x) - K \cdot r^{-t} \cdot N(x - \sigma\sqrt{t})$$

where

$$x = \frac{\ln \frac{S}{K \cdot r^{-t}}}{\sigma \cdot \sqrt{t}} + \frac{1}{2} \cdot \sigma \cdot \sqrt{t}$$

As a comment we would like to say that according to us a real advanced option system should use more sofisticated methods than Black & Sholes for evaluation of options as investigations show that some of the conditions for the model is not fulfilled on today's option market.

---

[13]Actually σ denotes the standard deviation, i.e. the square root of the volatility.

B–1.5 Positions

A position is a combination of stocks and options of different types and series. You can cathegorize the positions in the following groups:

- Naked positions
- Hedges
- Spreads
- Combinations

B–1.5.1 Naked Positions

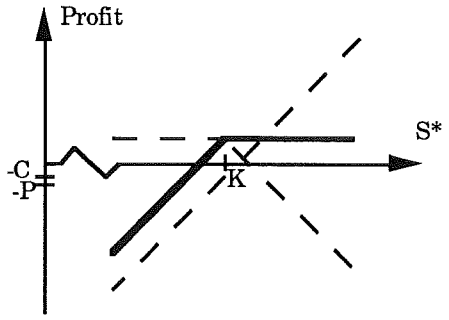
A position which only contains a stock or an option is called a naked position. The naked positions you can take are:

- Bought stock
- A blanked stock, i.e. a stock you have borrowed from someone and then sold in order to buy the stocks back later and return it to the person you have borrowed it from. As banks and dealers according to Swedish law are forbidden to take part in blanking it appears very seldom in Sweden.
- Bought call
- Written call
- Bought put
- Written put

B–1.5.2 Hedges

A Hedge is a position which consists of a stock and an option in such an order that one safeguards the other.

Example:
Buy one stock and write one call with the same striking price as the current stock price



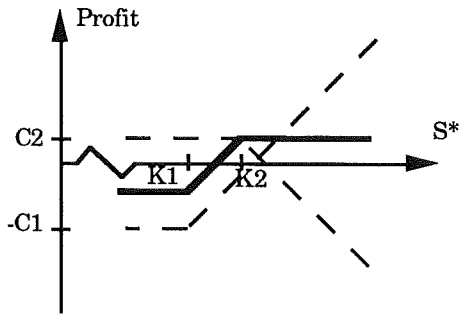
The bold line in the figure shows profit/loss for the position as a function of S*. This position is called 1:1 hedge.

B–1.5.3 Spreads

If you combine options of the same type but with different striking price or striking date where some are bought and others are written you get what is called a spread. A spread with options with the same striking date is called a vertical spread or a price spread. A spread

where on the other hand you let the stiking date vary while the striking price is the same is called a horizontal spread or time spread. Spreads with different striking price and striking dates are called diagonal spreads but these are quite unusual.

Example:
Buy a call with low striking price and write a call with high striking price.

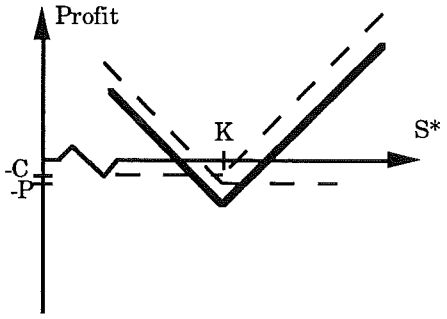


As the position above is a vertical spread and is profitable at a high stock price it is called hausse price spread. If you on the other hand buy the option with a high striking price and write the one with the low striking price you get a vertical spread which gives profit at a low stockprice and therefore is called baisse price spread.

B–1.5.4 Combinations

A combination is a position in which one has combined a call and a put with the same asset. They are either both bought or both written.

Example:
Buy a call and a put with the same striking price and striking price.



The position you get is called a straddle and gives profit if the stock price goes up or down a lot, but gives a loss if it is steady.

B–1.6 The actors on the market

The actors on the stock market can be cathegorized after which purpose they have in dealing with options. The cathegories are:

- Arbitragers
- Hedgers
- Traders

In real life you find few actors who act only in one rôle.

B–1.6.1 Arbitragers

An arbitrager is an actor who makes business in order to make money on options which are not accurately valuated. There are a lot of different ways of making money in this area but they are often quite complicated and out of the scope of this overview. Anyone can see, though, that if the whole range of the bold line in the diagrams above is above the S*-axis the position gives a profit independently of what the value of S* is.

The Arbitrager does not risk any money but his possible profits are usually very small.

B–1.6.2 Hedgers

The hedger uses options as an instrument to safeguard his portfolio against unfavourable changes on the market. An owner of a great portfolio of Volvo stocks who is afraid that there is a chance that the stock price might go down could, if he doesn't want to sell his shares, buy puts. If the stock price goes down he can exercise his options, if not, he lets the option expire. The options can in this case be seen as an insurance against a price fall.

B–1.6.3 Traders

It is easy to verify that a change of the price of a stock gives a relatively higher change in the options which have the stock as asset. This means that the trading with options gives an opportunity to higher profits (and losses) than the trading with shares. The groups of actors that trade on the option market due to this fact are called traders.

B–1.7 The generated system

We have made a option reasoning system using the knowledge described above. The user program supplies the reasoning machine with all options available, and then tells the machine what kind of actor – actors – it is who wants advice. The reasoning machine may then tell the user what positions the different users should take.

```
...
Expert :- new OptionExpert;
...
for ... all options available ... do
   Expert.OptionFact.NoteFact(Option);
...
OneActor :- new Trader;
AnotherActor :- new Hedger;
...
Expert.Actor.NoteFact(OneActor);
Expert.Actor.NoteFact(AnotherActor);
...
Expert.GoodPositions.print;
...
```

B–2 Sport Doctor

The other example we wrote was a small Sport Doctor. This example was made to test how our concept would work in a domain that wasn't built on mathematics and calculations but relied more on text-based and sometimes shallow knowledge. We chose Sport Doctor mainly due to the following reasons: Firstly, we had a lot of own experience from this area doing a lot of sports ourselves. Secondly we could easily get in touch with people who were experts on the subject and thirdly, the medical diagnosis area can be seen as one of the "classic" application areas for expert systems.

We have to admit that the generality of our system is more difficult to utilize in such an application as the Sport Doctor. But apart from a few minor things (which are described below) the overall impression is that it is quite easy to write even those kinds of systems that the generator was not originally designed for.

B–2.1 Some basic thoughts at the design of the system

Our objective was never to create a complete system but was to investigate methods, procedures, and other problems of this specific area. In order to make the system reasonably small but realistic in some sense we have limited it to only handle injuries in some parts of the leg, namely the ankle, the lower leg, and the knee. The system is still useful, though, because a great deal of the most common sport-injuries are located to these parts.

The system was, however, designed in such a way so that any extention would be as easy as possible. If anyone would like to enhance the system with more rules for the injuries already in the system, new injuries, or new body parts, it is very easy to do so.

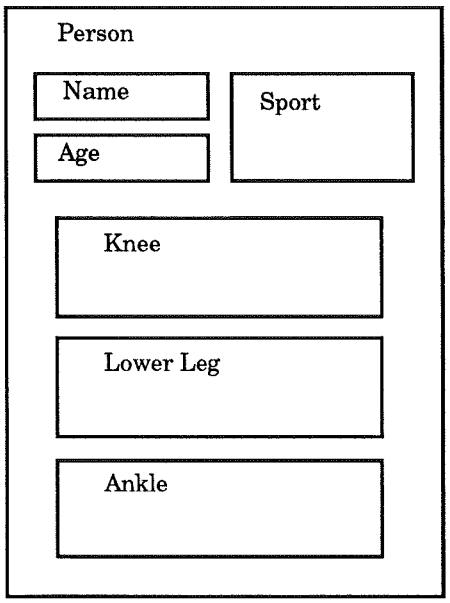
The system is built on interaction between the patient and the expert, where the expert puts a lot of different questions to the patient. (Most of them only demanding the answer yes or no.) When designing the system we have tried to follow two basic ideas about the questioning. First, the questions should come in a natural order from the patients point of view and secondly, the patient shouldn't have to answer more questions than necessary for the diagnosis. The use of the attribute classes described in appendix A prevents the system from asking questions until their answer is really necessary.

B–2.1.1 Object representation

In a Sport Doctor system it is natural to represent the patient as an object in the object oriented knowledge representation.This we have done by defining the class "Person". Unfortunately it is not possible to utilize the system's abilities to handle an arbitrary number of objects as a consultation usually is made by *one* patient with a statically allocated set of body parts. It is true that you could

imagine a consultation of a whole football-team at the same time – which is feasible in our system – but this would probably lead to a rather chaotic situation.

An advantage with the object oriented knowledge representation is however the fact that you can represent the anatomy of the athlete in the object. The class Person with its attributes is described in the figure below.



Person with its attributes

The two first attributes are name and age. The name is only used to give the object an identity, but the age attribute is used in a lot of rules as many injuries are tied to a certain age.

The attribute "Sport" is the sport that the person has been doing when he/she has been injured. This attribute is also used in a lot of rules since some injuries are typical for certain sports.

There are three attributes in the system which represent parts of the body, namely Knee, LowerLeg, and Ankle. These attributes are chosen since a lot of very common injuries in sports like running and soccer are located to those parts of the body. The principle of the division has been to make it as natural for the athlete as possible. (Usually an athlete knows that he has "pain in the lower leg", but no idea if its located to tibealis anterior or some other muscle.) Each part belongs to the class Enumerated Attribute (See Appendix A – Utilities) and the enumeration represents a finer division which is adjusted to the different diagnoses that are possible.

B–2.1.2 Rulerepresentation

The system is constructed as a backward chaining system where backward chaining is made on the fact diagnosis. Every the rules that describes a certain injury points at the fact diagnosis which is parameterized with the name of the injury. An example of such a rule is shown below.

```
## Rule Achilles Tendinitis
   #var
   ref(Person) P;
   ref(Diagnosis) D;

   #exists
   Achillestendon(P)
   TendinitisSymptom(P)

   #and
   BooleanQuestion("You feel pain when you start to run.")
or else
    BooleanQuestion("You feel pain when you walk on toes.")

        #action
        D:- new Diagnosis("Achilles tendinitis");
        #end
   #fact
   Diagnosis(D);
##end
```

Achilles Tendinitis Rule

As you can see a DiagnosisObject is created in the action statement with the name of the diagnosis as actual parameter. This object is then put in the fact of the rule.

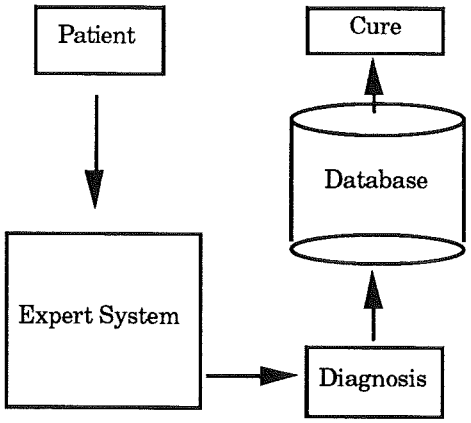
The procedure described above enables several different diagnoses to be made at one consultation. This is quite appropriate as a 'real' doctor often has to work with a number of hypotheses when the information is incomplete.

A large part of the work has been made to get a natural and effective question process. In order to spare the patient from irrelevant questions one has to gradually cut off parts of the rule tree. Some rules only serve this purpose and are used to classify the injuries into groups.

An objective with the structuring of the rules has also been to try to get a system which on the surface is adapted for the patient but internally works in a scientifical manner. The internal representation is therefore structured after medical terms while the textual part of the promts and questions is made to get an understandable user interface.

B–2.1.3 Possible enhancements

An extension that we already mentioned is to add more rules, injuries, and parts of the body into the system. Another enhancement which is natural is that the doctor apart from making diagnoses would be able to describe a cure for the injury. Such knowledge does not require a reasoning machine, and should be easy to handle by an ordinary database management system. The Sport Doctor would then produce diagnoses which were used as input to a database system which contained treatments and cures for different diseases and injuries as shown in below.



An extended SportsDoc

B–2.1.4 Execution example

This example demonstrates how a consultation of the Sport Doctor could look like.

```
$ sportsdocmain
What is the name of the Athlete?
> Bob
Which bodypart is hurted?
> knee
Were on the Knee does it hurt?
> ?
Choose between:
Whole joint
Inside
Outside
Below knee cap
Knee cap
Back

Were on the Knee does it hurt?
> Below knee cap
Which sport do you practise?
> running
How old are you ?
> 20
Is it true that: You feel pain in the knob below the knee
cap.
> y
Is it true that: You feel pain when you start running.
> y

• Diagnosis: Morbus Schlatter
```

References

General

[Birt79]
Birtwistle, G.
Dahl, O-J.
Myrhaug, B.
& Nygaard, K. Simula Begin. Studentlitteratur 1979.

[Clock81]
Clocksin, W.F.
Mellish, C.S. Programming in Prolog. Springer Verlag 1981

[Duda79]
Duda, R O.
Hart, P E.
Konolige, L.
& Reboh, R. A Computer-based Consultant for Mineral Exploration. Technical Report, SRI International sept 1979.

[Erik85]
Eriksson, Göran.
& Holm, Per. Programmering i Simula. DNA-LTH.

[Gold83]
Goldberg, Adele
& Robson, David Smalltalk-80: the language and its implementation. Addison-Wesley 1983.

[Hart78]
Hart, P E.
Duda, R O.
& Einaudi, M T. A Computer-based Consultation System for Mineral Exploration. Technical Report, SRI International may 1978.

[Nexp88] Nexpert Object. System Manual. Neuron Data 1988.

[Rich88]
Rich, Elaine. Artificial Intelligence, MacGraw-Hill 1988.

[Strou87]
Stroustrup, Bjarne — The C++ language. Addison-Wesley 1987.

[West88]
West, J. — Programming with Macintosh Programmers Workshop. Bantam Books 1988.

Options

[Cox85]
Cox, John
& Rubinstein, Mark. — Option Markets, Prentice-Hall 1985.

Sports injuries

[Petter77]
Petterson L.
& Renström P. — Skador inom Idrotten. Tidens förlag 1977.

[Read85]
Read, Malcolm — Idrottsskador. Libers förlag 1985.

[Snell81]
Snell, Richard S. — Clinical Anatomy for Medical Students. Little Brown and Company 1981.