Under Construction: Knowledge Based Solutions

by Bob Swart

Bolesian, the company for which I work (in Helmond, The Netherlands), specialises in Knowledge Based Solutions. This means that we create customised applications for our clients that perform tasks for which a certain amount of knowledge is required. Usually we use special knowledge engineering development systems like AionDS (from Platinum), but we also sometimes use Delphi or C++, especially where the focus is more on the user interface, or some other part of the engine, and less on the knowledge base itself.

In this month's column I would like to show how you can write a very simple knowledge based application using Delphi. Some kind of reasoning is applied, rather than simple mathematical functions. Usually, this is done in the form of *IF condition THEN conclusion* rules, but other techniques can be used as well.

Typically, Knowledge Based Systems consist of two parts: the knowledge base (with the IF-THEN rules) and the inference engine (the algorithm to reason with these rules). Hence, the knowledge is separated from the engine. This also means that we can't just implement the IF-THEN rules in plain Delphi code, since this would mean that they're just linked in with the engine code. The crux of a Knowledge Based System is that the knowledge rules can be maintained without having to recompile the system (since the system only consists of the inference engine which doesn't change, of course).Therefore, we can often identify a third component of Knowledge Based Systems: the rule editor, to insert, edit or update rules in the knowledge base (also called the rulebase). For now, we'll focus our attention on the knowledge base and inference engine only.

Figure 1 shows an example of a set of five IF-THEN rules that can be used to identify the well-known Delphi 1.x *"Error: disk full or file not an .EXE file"* error message problem. These five rules are in fact using 7 "facts" (issues that are reasoned about in the rules):

```
ERRORMESSAGE (the goal)
DISK-IS-FULL
USING-DELPHI-1
OPTIMISE-FOR-SIZE/LOAD-TIME-CHECKED
CORRUPT-OR-WRONG-RES-FILE
32-BITS-RES-FILE
USING-IMAGEEDITOR
```

The *goal* is what we need to "prove" or "explain". In this very simple example, facts are limited to having only three values: Unknown (initial default value), Yes or No. When using any of the rules above we can only derive a value Yes for a fact. However, if a fact cannot be derived from a rule (like the fact `USING-DELPHI-1`), then we have to obtain its value some other way, usually by asking the user (in a nicely formatted question of course). Also, asking fact values should always be the last resort and only used if the fact cannot be derived using one of the rules. Some facts can have a pre-defined initial value other than Unknown (for example if we have a "user profile" which tells us the user is using Delphi 1.x). So, in the correct order, a fact can obtain a value in three ways:

- Initially (default Unknown, but sometimes overruled to Yes or No),
- Derived from a rule (Yes),
- Obtained from the user by asking for it (Yes/No).

Forward Chaining

Having these facts together with the rules, we can use (at least) two algorithms to apply the rules on the facts. The simplest algorithm is known as *forward chaining*. This means that given a few facts with known values, we attempt to apply each rule until we have either found a valid goal (in this case the `ERRORMESSAGE`), or we've run out of rules that can be executed. Note that we need to do this in loops, as the execution of one rule can allow another rule to be executed (which may have been disregarded a moment ago). In pseudo code, this looks like Listing 1.

The function `TestRule` checks if the conditions of the rule are valid.

➤ *Figure 1*

```
IF DISK-IS-FULL THEN ERRORMESSAGE
IF USING-DELPHI-1 AND
   OPTIMISE-FOR-SIZE/LOAD-TIME-CHECKED THEN ERRORMESSAGE
IF CORRUPT-OR-WRONG-RES-FILE THEN ERRORMESSAGE
IF USING-DELPHI-1 AND
   32-BITS-RES-FILE THEN CORRUPT-OR-WRONG-RES-FILE
IF USING-IMAGEEDITOR THEN CORRUPT-OR-WRONG-RES-FILE
```

➤ *Listing 1*

```
repeat
  rule := first;
  repeat
    if TestRule(rule) then FireRule(rule);
    Next(rule);
  until goal found or no more rules;
until goal found
```

The procedure FireRule executes the conclusion of the rule, which means assigning a Yes value to one (or more) of the facts.

Forward chaining is especially useful if you have a lot of facts and goals and do not know in advance which of the goals can be derived. In our little example, however, we have only one goal that we need to prove or explain, so we need to go to another inference algorithm called *backward chaining*.

Backward Chaining

Backward chaining starts with the goal we want to prove and looks for rules that have this goal in their conclusion part. These rules are then checked to see if they can be executed. If not (which is usually the case), we attempt to prove the conditions of the rule after which the rule can be fired, which then leads to the required goal. This means recursively attempting to execute rules until enough have been successfully executed to allow the goal to be proved. Listing 2 shows this in pseudo code.

Although this procedure is recursive, we can predict in advance what the maximum depth will be (in our example there are two layers of rules, so the depth, including the initial goal, is three).

Facts

In order to implement a knowledge based system in Delphi, we need some way to store the facts and rules. It seems logical to use tables for this purpose. For a fact, we need to store a unique fact number (the key), whether or not it's a goal-fact (Boolean), the name of the fact, and (if appropriate) a question that can be asked if the fact cannot be derived using rules.

This leads to the table FACTS.DB shown in Figure 2.

Note that we have only one goal-fact and five facts that (if rules fail) we can get a value for by asking a

➤ *Listing 2*

```
procedure backward(goal)
begin
  rule := first;
  repeat
    if Conclusion(rule,goal) then
    begin
      if not TestRule(rule) then { derive }
        for each condition in rule call backward(condition)  // recursive call
      if not TestRule(rule) then { ask }
        ask fact for condition(s) of rule
    end
    Next(rule)
  until goal proven or no more rules
end
```

➤ *Listing 3*

```
unit FACTS;
interface
uses DB, DBTables, SysUtils;
Type
  TName32 = String[32];
  TValue = (UnKnown, Yes, No);
var
  ValueStr: Array[TValue] of String[7] =
    ('UnKnown','Yes','No');
type
  TFact = class(TObject)
  private
    FFact: Integer;
    FGoal: Boolean;
    FName: TName32;
    FValue: TValue;
    FQuestion: ShortString;
  public
    constructor Create(Table: TTable); virtual;
  published
    property Fact: Integer read FFact;
    property Goal: Boolean read FGoal;
    property Name: TName32 read FName;
    property Value: TValue read FValue write FValue;
    property Question: ShortString read FQuestion;
  end {TFact};
const
  MaxFact = 512; { limit to 512 facts for now }
var
  NumFact: Integer = 0;
  { optimisation: fact # in position # }
  _Fact: Array[1..MaxFact] of TFact;
implementation

constructor TFact.Create(Table: TTable);
begin
  inherited Create;
  with Table do begin
    FFact := FieldByName('Fact').AsInteger;
    FGoal := FieldByName('Goal').AsBoolean;
    FName := FieldByName('Name').AsString;
    FValue := UnKnown;
    FQuestion := FieldByName('Question').AsString
  end
end {Create};

const FactTableName = 'FACTS.DB';

procedure CreateFACTS;
begin
  with TTable.Create(nil) do
  try
    Active := False;
    TableType := ttParadox;
    TableName := FactTableName;
    with FieldDefs do begin
      Clear;
      Add('Fact', ftInteger, 0, TRUE);
      Add('Goal', ftBoolean, 0, TRUE);
      Add('Name', ftString, 32, TRUE);
      Add('Question', ftString, 255, FALSE)
    end;
    with IndexDefs do begin
      Clear;
      Add('index', 'Fact', [ixPrimary,ixUnique])
    end;
    CreateTable
  finally
    Free
  end
end {CreateFACTS};

var FactTable: TTable = nil;

initialization
  if not FileExists(FactTableName) then CreateFACTS;
  FactTable := TTable.Create(nil);
  FactTable.TableName := FactTableName;
  FactTable.Open;
  while not FactTable.Eof do begin
    Inc(NumFact);
    if FactTable.FieldByName('Fact').AsInteger <>
      NumFact then
      raise Exception.Create(
        'Error: facts are not sorted...');
    _Fact[NumFact] := TFact.Create(FactTable);
    FactTable.Next
  end;
finalization
  FactTable.Close;
  FactTable.Free
end.
```

Table : Facts.db

Facts	Fact	Goal	Name	Question
1	1	True	Disk Full or file not a EXE file	
2	2	False	Disk Full	Is your disk really full?
3	3	False	Optimise for size/load time	Do you have the "optimise for size/load time" option set in the IDE?
4	4	False	Corrupt or wrong .RES file	
5	5	False	Using Delphi 1.0x	Are you using Delphi 1.0x?
6	6	False	32-bits .RES file	Did you use a 32-bits .RES file (with Delphi 1.0x)?
7	7	False	ImageEditor	Did you use ImageEditor to create this .RES file?

➤ *Figure 2*

Table : Rules.db

Rules	Rule	CF	Fact	Value	Comments
1	1	0	2	1	if your disk is indeed full then...
2	1	1	1	1	Error: Disk full or not a valid .EXE file message
3	2	0	3	1	if you've checked the optimise for size/load time option and
4	2	0	5	1	if you are using Delphi 1.0x then
5	2	1	1	1	Error: Disk full or not a valid .EXE file message
6	3	0	4	1	if you have a corrupt (or wrong version of your) .RES file then
7	3	1	1	1	Error: Disk full or not a valid .EXE file message
8	4	0	5	1	if you are using Delphi 1.0x and
9	4	0	6	1	if you're using a 32-bits .RES file then
10	4	1	4	1	you have a wrong version of the .RES file
11	5	0	7	1	if you've used ImageEditor to create your .RES file then
12	5	1	4	1	you can get a corrupt .RES file

➤ *Figure 3*

question. Also note that the value of the facts is not stored in the table. Reasoning is a process that is repeated several times for different situations, so it would not be useful to store the values for one particular session in the table. Instead, the table is loaded in a set of fact classes and values are stored in memory with the fact classes.

The corresponding class `TFact` is defined in Listing 3.

Note that if the table FACTS.DB does not exist, this program will automatically create it (this code was generated using my Table Source Expert available from my Web site). If the table does exist, this unit reads the facts from the table and inserts them in an array `_Fact of TFact`. This is the place where the actual value (default Unknown) of each fact is stored and can be changed during the reasoning process. Note that for the time being I've used a hardcoded limit of 512 facts. This is more than enough for our current example. Next time, we'll see how to store a virtually unlimited number of facts in a much cleaner way.

Finally, note that the fact table must not contain any 'holes', that is, facts must be numbered from 1 to N where N is the number of facts in the table. Since the table is sorted on fact-number, this would mean that record X is also fact number X. This is very important and speeds up the process of referencing facts by rules. If this condition is not true, then an exception *Error: facts are not sorted...* will be raised.

Rules

A table similar to FACTS.DB can be written for the rules. Rules consist of two parts: conditions and conclusions. Both can be multi-valued. A condition consists of a rule number, a fact (key-number) and the value that the fact must have to satisfy the condition. A conclusion also consists of a rule number, a fact (key-number) and the value that is assigned to this fact once the conditions are satisfied and the rule is fired. A conclusion can also contain a certainty factor (CF), usually a floating point value between 0 and 1 that indicates the certainty of the conclusion. We can even combine certainty factors with complex mathematical rules, but for now we only use a CF of 1.

If we read the previous paragraph closely, we see that the conditions and conclusions are in fact very much alike. The only thing that conclusions have is a CF of 1. So, let's use this fact to create a single record type where conditions are defined to have a CF of 0 (this would be a useless conclusion anyway). This leads to the table RULES.DB shown in Figure 3.

Note that I've introduced some comments for each rule, to explain what the rule actually is about. This can be used later for explaining the reasoning process (something we'll explore in more detail next month). The corresponding class `TRule` is shown in Listing 4.

Note that just like the facts, the RULES.DB table is created when it doesn't already exist. When it does exist, the rules are read into the `_Rule` array (fixed length of 1024 rules for now). Apart from the `TRule` class, we need three more routines to enable us to reason with the rules:

```
function TestRule(
  RuleNr: Integer): Boolean;
procedure FireRule(
  RuleNr: Integer);
function Conclude(RuleNr,
  FactNr: Integer): Boolean;
```

Function `TestRule` returns `True` if the conditions of the rule `RuleNr` are satisfied. In that case, we can safely call `FireRule`, which executes the conclusions of rule `RuleNr`, which means that certain facts get a new value. The last routine, `Conclude`, is needed to find out if the given rule `RuleNr` does in fact contain a conclusion part that (amongst other possibilities) derives something for fact `FactNr`. This last function is needed for backward chaining, to find out which rules to use to prove a fact.

Forward Chaining

Now that we have real `TFact` and `TRule` classes, it's time to start implementing the forward and backward chaining inference mechanisms. Forward chaining is fairly straightforward, as you can see in the pseudo-code. The real Delphi code is shown in Listing 5.

Of course, if none of the facts have any initial value this loop will run once and then terminate without having found a goal. In a situation with only one goal, forward chaining is not the best algorithm to use (as we discussed earlier).

Backward Chaining

While forward chaining has an upper limit of N execution times of the loop (for N rules, since in each loop a minimum of one rule must be executed to avoid termination), backward chaining is a recursive process of a maximum depth of N (if each rule is needed to prove the next one). In theory, backward chaining can even lead to endless loops (until we run out of stack space) in cases like this:

```
IF A THEN B
IF B THEN A
```

where we need to derive A. One way to avoid that would be to mark each rule for which we're busy right now (a kind of transitive closure). For our small example, this technique was not needed, but we'll get back to this next month when we'll enhance TRule and create a real TRuleBase component.

The pseudo-code for backward chaining can be translated into the real Delphi code shown in Listing 6.

Note that if a question has been answered with No, we use a fast exit

➤ *Listing 5*

```
function Forwards: Integer;
var RulesFired,i: Integer;
begin
  Result := 0;
  RulesFired := NumRule;
  while (Result = 0) and (RulesFired > 0) do begin
    RulesFired := 0;
    for i:=1 to RuleMax do begin
      { all rules }
      if TestRule(i) then begin
        FireRule(i);
        Inc(RulesFired)
      end
    end;
    Result := NumFact;
    while (Result > 0) and ((not _Fact[Result].Goal) or
     ((_Fact[Result].Goal) and (_Fact[Result].Value = UnKnown))) do
     Dec(Result)
  end
end {Forwards};
```

➤ *Listing 4*

```
unit RULES;
interface
uses DB, DBTables, SysUtils, FACTS;
type
  TRule = class(TObject)
  private
    FRule: Integer;
    FCF:  SmallInt;
    FFact: Integer;
    FValue: TValue;
    FComments: ShortString;
  protected
    FFired: Boolean;
  public
    constructor Create(Table: TTable); virtual;
  published
    property Rule: Integer read FRule;
    property CF:  SmallInt read FCF;
    property Fact: Integer read FFact;
    property Value: TValue read FValue;
    property Fired: Boolean read FFired write FFired;
    property Comments: ShortString read FComments;
  end {TRule};
const MaxRule = 1024;
var
  NumRule: Integer = 0;
  RuleMax: Integer = 0;
  _Rule: Array[1..MaxRule] of TRule;
function TestRule(RuleNr: Integer): Boolean;
procedure FireRule(RuleNr: Integer);
function Conclude(RuleNr, FactNr: Integer): Boolean;
implementation
constructor TRule.Create(Table: TTable);
begin
  inherited Create;
  with Table do begin
    FRule := FieldByName('Rule').AsInteger;
    FCF := FieldByName('CF').AsInteger;
    FFact := FieldByName('Fact').AsInteger;
    FValue := TValue(FieldByName('Value').AsInteger); // 0,1,2
    FComments := FieldByName('Comments').AsString
  end
end {Create};
function TestRule(RuleNr: Integer): Boolean;
var i: Integer;
begin
  Result := True;
  for i:=1 to NumRule do
    if (_Rule[i].Rule = RuleNr) and (_Rule[i].CF = 0) then
      Result := Result AND
        (_Fact[_Rule[i].Fact].Value = Rule[i].Value)
end {TestRule};
procedure FireRule(RuleNr: Integer);
var i: Integer;
begin
  for i:=1 to NumRule do
    if (_Rule[i].Rule = RuleNr) and
       (_Rule[i].CF > 0) and not _Rule[i].Fired then begin
       { fire }
       _Fact[_Rule[i].Fact].Value := _Rule[i].Value;
        _Rule[i].FFired := True
    end
end {FireRule};
function Conclude(RuleNr, FactNr: Integer): Boolean;
var i: Integer;
begin
  Result := False;
  for i:=1 to NumRule do
    if (_Rule[i].Rule = RuleNr) and
      (_Rule[i].Fact = FactNr) and
      (_Rule[i].CF > 0) then Result := True { rule can be used }
end {Conclude};
const RuleTableName = 'RULES.DB';
procedure CreateRULES;
begin
  with TTable.Create(nil) do
  try
    Active := False;
    TableType := ttParadox;
    TableName := RuleTableName;
    with FieldDefs do begin
      Clear;
      Add('Rule', ftInteger, 0, TRUE);
      Add('CF',  ftSmallInt, 0, TRUE);
      Add('Fact', ftInteger, 0, TRUE);
      Add('Value', ftInteger, 0, FALSE);
      Add('Comments', ftString, 255, FALSE)
    end;
    with IndexDefs do begin
      Clear;
      Add('index', 'Rule;CF;Fact', [ixPrimary,ixUnique])
    end;
    CreateTable
  finally
    Free
  end
end {CreateRULES};
var RuleTable: TTable = nil;
initialization
  if not FileExists(RuleTableName) then CreateRULES;
  RuleTable := TTable.Create(nil);
  RuleTable.TableName := RuleTableName;
  RuleTable.Open;
  while not RuleTable.Eof do begin
    Inc(NumRule);
    _Rule[NumRule] := TRule.Create(RuleTable);
    if _Rule[NumRule].Rule > RuleMax then
      RuleMax := _Rule[NumRule].Rule;
    RuleTable.Next
  end;
finalization
  RuleTable.Close;
  RuleTable.Free
end.
```

of the loop (by setting `j` to `NumRule` and ignoring all other conditions of the current rule which is under investigation). In our example, we can only use conditions that are `True` (value of `Yes`), so whenever we get a `No`, we can disregard the rule immediately. Of course, in the real world we often need to check a fact against a real value, or even a range of values, and this short-cut cannot be used in those situations.

Action!

Now, let's see the backward chainer in action. In order to prove the goal-fact 1, it tries to find a rule that has fact 1 as a conclusion. There are three rules than can be fired: 1, 2 and 3. Rule 1 can be fired if the fact `Disk Full` can be proven. However, there is no rule to prove this fact, so we have to ask a question (`Is your disk really full?`) and await the answer.

We answer `No`, so the inference engine discards rule 1 and moves on to the next one that can be used to prove fact 1. This turns out to be rule 2, which can be fired if the fact `Optimise for size/load time` can be proved. Again, there is no rule to prove this fact, so we ask `Do you have the 'optimise for size/load time' option set in the IDE?`

We answer `No` and move on to the next rule. Rule 3 says that the goal-fact 1 can be proven if we can prove fact 4: namely that we're using a corrupt or wrong .RES file. There are two rules that can be used to prove fact 4, rules 4 and 5. The first rule consists of two conditions: `Using Delphi 1` and `Using a 32-bits RES file (with Delphi 1)`. Neither fact can be proven using rules, so we need to ask `Are you using Delphi 1.0x?`.

Yes, we are using Delphi 1.0x, so the program continues by trying to prove the second condition: we were using a 32-bits .RES file. Well, no, we didn't use a 32-bits .RES file. This means that one condition is proven but the other one is not. Rule 4 must be disregarded and we move on to rule 5. This rule can prove fact 4 if we used the Image Editor. This fact cannot be proven, so we ask `Did you use ImageEditor to create this .RES file?`.

```
procedure Backwards(Goal: Integer);
var
  i,j: Integer;
begin
  i := 1;
  while i <= RuleMax do begin
    { all rules }
    if Conclude(i,Goal) then begin
      if TestRule(i) then
        FireRule(i)
      else begin
        { infer or ask }
        j := 1;
        while j <= NumRule do begin
          if (_Rule[j].Rule = i) and (_Rule[j].CF = 0) and
            (_Fact[_Rule[j].Fact].Value = UnKnown) then begin
            Backwards(_Rule[j].Fact); { infer }
            if TestRule(i) then
              j := NumRule
            else begin
              { ask }
              if _Fact[_Rule[j].Fact].Question <> '' then begin
                if MessageDlg(_Fact[_Rule[j].Fact].Question,
                  mtConfirmation,[mbYes,mbNo],0) = mrYes then
                  _Fact[_Rule[j].Fact].Value := Yes
                else begin
                  _Fact[_Rule[j].Fact].Value := No;
                  j := NumRule { rule can never be fired }
                end
              end;
              if TestRule(i) then
                j := NumRule
            end
          end;
          Inc(j)
        end;
        if TestRule(i) then begin
          FireRule(i);
          i := RuleMax
        end
      end
    end;
    Inc(i)
  end
end {Backwards};
```

➤ *Listing 6*

Let's assume I did use the Image Editor to create a .RES file. The inference engine can use this proven fact to fire rule 5 which then proves the fact of a corrupt or wrong .RES file (this would a good place to use a real certainly factor, by the way, since the Image Editor is known to corrupt .RES files in only a few rare circumstances, however, enough to check it out in this case). Having proven that we have a corrupt .RES file, we can then fire rule 3 and prove (explain) our goal-fact 1.

Having found a closing path to prove the goal-fact, we can now print all rules that were fired and all the facts that turned out to be true (including the comments to give extra information). This enhancement will be implemented next month. For now, we at least know what knowledge based solutions are, how they work and how we can implement a very simple example using Delphi 2.

Next Time

I've already moved several 'advanced' issues to next month. One of them is the issue that the current `TFact` and `TRule` classes are not real components but derived from `TObject`. We'll fix that next time and include some component editors to edit the facts and rules while we're at it. We'll also be extending the facts to include ranges of values, optimise the backward chaining process and include an explanation facility: the so-called "why" function that explains to the user why (or how) a conclusion has been reached.

Bob Swart (aka Dr.Bob, http://home.pi.net/~drbob/) is a professional knowledge engineer specialist using Delphi and C++ for Bolesian, a free-lance technical author for The Delphi Magazine and co-author of *The Revolutionary Guide to Delphi 2*. In his spare time, Bob likes to watch video tapes of Star Trek Voyager and Deep Space Nine with his 2.5 year old son Erik Mark Pascal and newborn daughter Natasha Louise Delphine.