Available online at www.sciencedirect.com

SCIENCE DIRECT®

Journal of Electron Spectroscopy and Related Phenomena 133 (2003) 87–101

JOURNAL OF ELECTRON SPECTROSCOPY and Related Phenomena

www.elsevier.com/locate/elspec

The "carbon contamination" rule set implemented in an "embedded expert system"

János Végh*

MTA ATOMKI, P.O. Box 51, H-4001 Debrecen, Hungary

Received 17 January 2003; received in revised form 28 August 2003; accepted 2 September 2003

Abstract

Different external expert system shells have been used as the basis for previous attempts to develop an expert system for the X-ray photoelectron spectroscopy (XPS). The present paper describes a reasoning expert system engine, which can be built directly into XPS data-acquisition and data-evaluation software. The feasibility of the realized system is demonstrated through implementation of a real-life rule set (the carbon contamination rules).
© 2003 Elsevier B.V. All rights reserved.

Keywords: Carbon contamination; Embedded expert system; X-ray photoelectron spectroscopy (XPS)

1. Introduction

Expert systems have been developed for a variety of applications, and Castle and Baker [1] have proposed a design for an expert system suitable for application in X-ray photoelectron spectroscopy (XPS). A workshop was held in St. Malo [2] in April 2002 to further develop the concept and rule base for this latter application. In the former expert system applications, some kind of expert system shell is used and the user communicates with it via a natural language; i.e. *the user* "interfaces" with the expert system by providing measurement and evaluation data, sample information, etc. The expert system shells are powerful and require extensive computer resources. However, most of their functionality is not necessary for the intended XPS-specific system. In addition, the presence of the user (with more or less expertise in the field) makes the process more or less subjective and cumbersome. In that form, it is not suitable for building an expert system into data-acquisition and data-evaluation software.

Another approach is suggested here. The most important part of the expert system: the "inference engine" is retained, and this is implemented in a way to allow access to the spectrum measurement, evaluation, sample handling, etc. data. A special "generic rule" form can then be constructed. Based on this, a special ('hard-wired') rule set can be designed, that applies logical operations to the replies from the informing objects. The system constructed in this way is able to answer questions from the user (using the available information sources) and is able to explain its decision. The forms in which questions are received and answers delivered are left open to allow maximum flexibility.

* Tel.: +36-52-417266; fax: +36-52-416181.
E-mail address: J.Vegh@atomki.hu (J. Végh).

0368-2048/$ – see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/j.elspec.2003.09.002

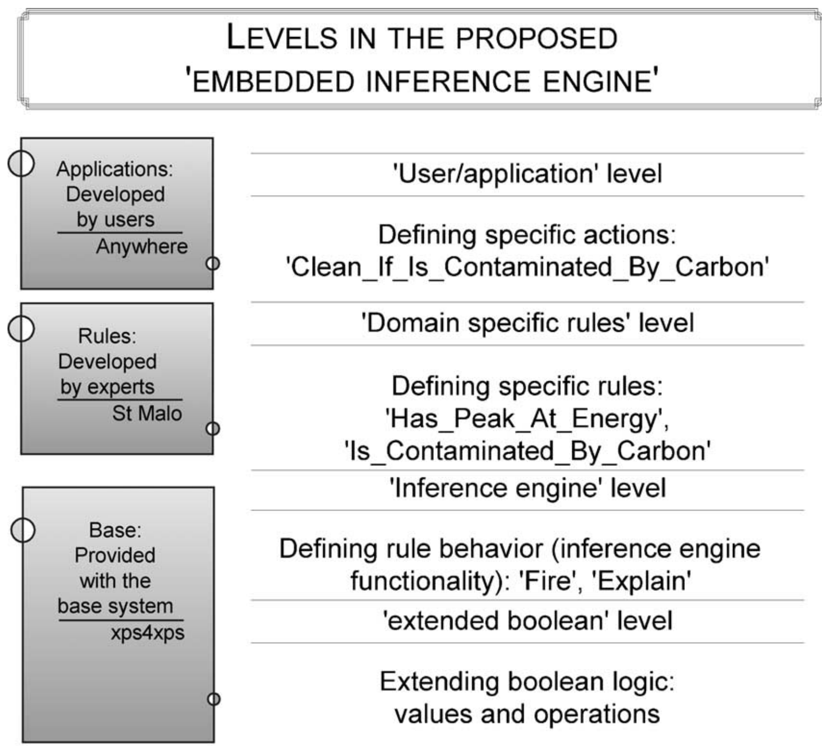

Fig. 1. Usage levels in the proposed system.

2. The 'embedded inference engine'

The suggested system assumes that the rules for the expert system can be constructed using pure logical functions. This system has three levels (see Fig. 1). On the base level ("Base"), the extended logic is defined (see Section 2.1), with operations. Based on this logic, a generic rule is constructed (see Section 2.3) to communicate with the informing objects and to assemble replies (including reasoning of the decision); i.e. it provides the functionality of the 'inference engine'. On the expert level ("Rules"), domain-specific rules are created that read the actual status of the comprised objects (for details see Fig. 2 and Section 3.3) and produce the established, expert-proven replies. On the user level ("Applications"), these decisions can be used or their reasoning can be studied.

For the first level (the first version of) a turnkey system is provided. On the expert level, a domain-specific XPS rule system is provided, and the "carbon contamination" rule set [1] is implemented as an example. It is hoped to add several more rules later. The user level remains completely within the user's software; as an example a demonstration for using the "carbon contamination" rule set is presented [3].

2.1. Extending Boolean logic

In real life the replies of an expert person to a question cannot only be a definite "yes" or "no", but even "maybe yes" or "probably no". It might also happen that some information is not available or is uncertain or is unknown. In addition, an expert person might say, "in lack of, . . . , I cannot decide", "I am not sure", or even "I do not know". These replies can be used as input information when formulating another question to an expert person. Obviously, a "maybe yes" is not identical with "yes". One of the most critical points of an expert system is the type of the output it can give and the type of the input it can receive.

For operating the inference engine properly, one has to extend the generally used (two valued) Boolean

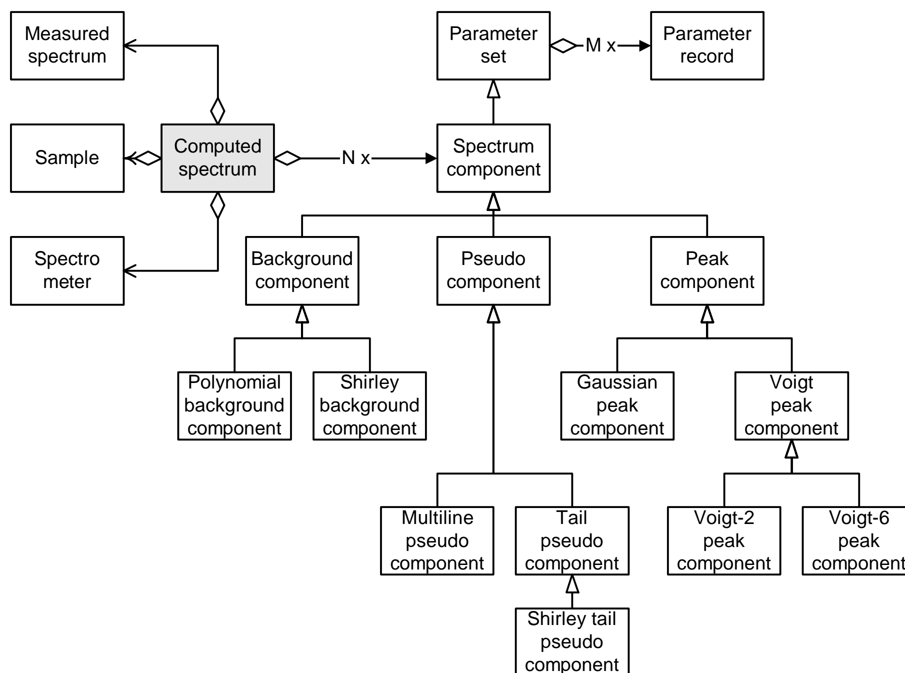

Fig. 2. Object interdependence in the expert system-controlled data-evaluation software.

logic to multiple-valued logic. The idea is not strange at all: see fuzzy logic and its applications, for example [4]. Obviously, it would be useless to extend the "yes/no world" with all the mentioned reply types; rather, introducing a third state ("unknown/not set/do not know") would suffice. Even this idea is not new: see [5]. Also note that the ANSII standard for SQL database-handling language (ANSI SQL 89) applies a triple-valued logic [6]. Introducing this extension enables the expert system to simulate an expert person who is able to deal with incomplete or not fully reliable input data, and is able to give a reply other than a definite "yes" or "no".

The new logic can be constructed as an extension to the Boolean logic: operators (defined by their "truth table", see Appendix A.1) can be defined and implemented. As shown in tables with shaded background, the two-valued Boolean logic is included as a subset of the newly introduced three-valued logic: this subset of truth tables is exactly equivalent to truth tables for two-valued logic. The "promotion" operation (mixing three-valued and two-valued logical variables) is also possible: operations between two-valued and three-valued Boolean values can be interpreted; their result is as shown in the table in Appendix A.

2.2. Introducing objects

For a complex task such as implementing an expert system, using Object Oriented Programming (OOP) seems to be a necessity. For non-programmers, the term "object" sounds rather mystic. Probably the best way to understand the term is via the "Disney-metaphor", as shown in Fig. 3. According to this, the "cleaning objects", which usually comprise only data, such as volume, length, weight, etc., are personalized and can also perform actions such as fill, wipe, clean, etc. In this way, *how* the action is carried out remains hidden; the "wizard" asks for some action (*what*) and the "cleaning objects" carry it out. The unnecessary details (*how*) are *encapsulated* into the objects, allowing for a better review of the task.

New objects can be derived from an object; the new object *inherits* all features and data from its *ancestor* and makes something more (*what*) or the same thing in a different way (*how*). For a more detailed (and professionally correct) discussion, please refer to the

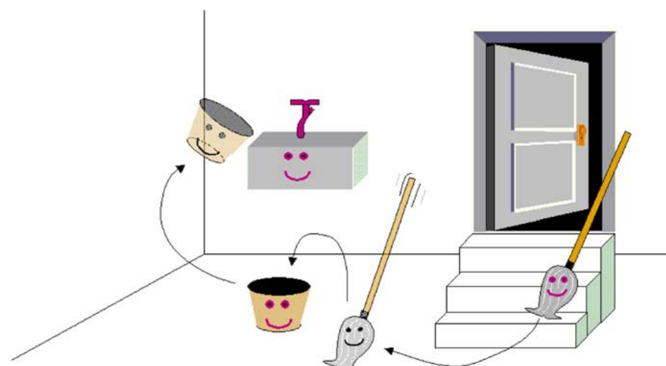

Fig. 3. The 'Disney metaphor' for objects.

numerous textbooks on object-oriented programming, for example [7].

2.3. The 'generic rule'

The 'rules' can be represented in a convenient way as objects; some of the important data and function names are shown in Appendix A.2.1. The data of the objects are a three-valued logical value and text strings; furthermore the methods are functions (comprising operators, logical functions and other computational methods). These rules have 'their own' value, which is calculated from some other rules and/or from some (properly communicated) external values. Since the resulting rules are inherited from extended Boolean logical functions, they are usable in rules as described in Appendix A.2.2.

In most of the known "expert system shells" a special language is constructed to describe the rules. In the present system only one special method is used which returns a rule. Otherwise, the language's standard elements are used to combine the rules, to calculate, etc. The present expert system is to be built into data handling software, so it shall be somewhat simplified, but still able to simulate a real domain expert. Since some well defined, community verified rules are needed for the intended application, it is an acceptable compromise to omit the native language input and only use the inferencing ability of the system.

The 'inference engine' functionality is hidden in the operator functions that take logical variables or rules as parameters and provide *generic rules* as results. In addition to performing the logical operation, these operator functions make "history notes" about the actual state of the used arguments. When asking for reasoning behind the resulting value, this history string (preceded by the name and the actual value of the rule) is returned. Although the built-in "default reasoning" is appropriate for most purposes, also "custom reasoning" is possible; when constructing rules, users can follow their own method of reasoning.

Note that two different evaluation modes can be set for the rules. In the "complete" mode, all logical variables and functions comprising the rules are evaluated, independently from the actual values the rules deliver; in "shortcut" mode only those from which the result of the rule can be clearly evaluated. For example, if the rule is "A **and** B", and the value of A is **false**, the result of the logical expression will be **false**, independently of the value of B. Because of this, in shortcut mode B will not be evaluated at all. Since the variables and functions that are not actually used for calculating the value of a rule, are redundant, omitting them makes the evaluation shorter and quicker as well as the reasoning cleaner. Because of this, the "shortcut" mode is selected as default.

2.4. Combining rules

Although rules are expressions, resulting in a logical value, any kind of operation can be used. In this way,

different kinds of combined rules can be constructed. In the examples below, meta-language expressions are shown.

In the case of a sub-range type rule, it is possible to provide alternative rules for the same goal, i.e. the sub-rules reply to the same question, but they are able to deliver this reply under slightly different conditions. The range of the comprised individual rules is limited, but the resulting rule covers the full range, thanks to the appropriate combination.

```
xrIsInRange(ThresholdLow,
  ThresholdHigh) =
{  xrIsInSubRange1(ThresholdLow,
  Val1)
    OR
    ...
    xrIsInSubRangeN(Val2,
      ThresholdHigh)
}
```

i.e. the sub-rules have a range of validity, and they “fire” only in case the conditions lie in their particular range of validity, otherwise reply with “I do not know”, thus allowing other sub-rules to decide. The rule then can be a simple “OR” of these functions. Another example is the “Multiple condition” rule, used by Castle and Baker [1].

```
xrMultipleCondition(ThresholdLow,
  ThresholdHigh) =
{  Percent = 0;
    if(Condition1 is True)     Percent
      = Percent + Percent1;
    ...
    if(ConditionN is True)     Percent
      = Percent + PercentN;
    if(Percent<ThresholdLow)
        return False;
    else if(Percent>ThresholdHigh)
        return True;
    else return Unknown;
}
```

Here, the different conditions contribute different amounts of certainty to decide the value of the rule. Finally the contributions are summed up. The rule results in **False** if the sum is below the lower threshold value, **True** if above the upper threshold value, and **Unknown** between them.

Note: If the result is “not True” then this is not the same as “False”!

Any other combination of rules can be easily assembled. Presently, the base system contains only pre-defined combination of rules (i.e. is hard-wired), which corresponds to the expert-proven rules sets.

3. Domain-specific rules

The functionality of the “generic rule” object provides a simple “inference engine” that needs “expert rules” as fuel. From the generic rules, *domain-specific rules* (for example generally valid in the field of XPS) can be derived. Based on these rules, more specific expert rules can be constructed.

3.1. How to code rules

The base packet (the inference engine) offers a rather complete functionality, so when making new rules, only the name of the new rule (“Name”) and the method which calculates the value of the rule (“Fires”) shall be provided, the rest will be done by the inherited functionality. A simple example is the rule, which results in the information if the binding energy is available.

In some cases the spectrum data points are recorded on a binding energy scale, in some cases on a kinetic energy scale and in some cases the scale type is not recorded at all in the data file. It might also happen that although the scale type and energy values are known, the value of the excitation energy is not recorded in the data file. (Of course, the experimenter knows it, but the embedded expert system cannot consult him.) In the case one needs an energy data on the binding energy scale, one has to know if it is available. In everyday language:

Binding energy is available
if
  energy data are on binding energy scale
    or
    energy data are on kinetic energy scale
      and
    excitation energy is known.

Using the present system, the only special method that shall be written for coding this rule is the ‘`Fire`’

method. The meta-language formulation of the rule is presented in Section 3.4, while the complete, working code (in C++) for this rule is shown in Appendix A.2.2.

As it is seen, it is quite straightforward to translate the everyday terminology to logical expression; the brackets in this particular case are not really necessary (the precedence of the operators would result in the same execution order without brackets, too), they just emphasize how exactly the expression is meant. The resulting rule is completely specified: now (thanks to the underlying object's functionality) the "`CalculateValue`" member function can calculate the resulting value (whether one can use the binding energy scale) and the method "`GetReasoning`" can tell the reason to the user.

Depending on the actual values of the statuses used for evaluating the rule, the "`Fire`" function delivers the correct value for the rule and the 'reasoning' function will result in text something like:

```
"Binding energy known" is TRUE
  because
"Energy-Data are BE" is TRUE
```

or

```
"Binding energy known" is TRUE
  because
("Energy-Data are KE" is TRUE
AND
"Exciting energy known" is TRUE)
```

or

```
"Binding energy known" is FALSE
  because
("Energy-Data are BE" is FALSE
OR
"Energy-Data are KE" is FALSE)
```

or

```
"Binding energy known" is UNKNOWN
  because
("Energy-Data are BE" is FALSE
OR
("Energy-Data are KE" is TRUE
AND
"Exciting energy known" is UNKNOWN))
```

Again, these "reasons" are from the working code, in the default "shortcut" evaluation mode. In the "complete" evaluation mode, the first example sounds:

```
"Binding energy known" is TRUE
  because
("Energy-Data are BE" is TRUE
OR
("Energy-Data are KE" is FALSE
AND
"Exciting energy known" is UNKNOWN))
```

As it is seen, the information content is identical but the output is more verbose. In the shortcut mode it is simpler to understand the reasoning of the rule.

3.2. Using external information

Unlike the example above, which uses other rules to calculate the value of the rule, most rules need data from some external information source like databases, data-evaluation program, spectrum-attached information, user, etc. The base package defines the generic object `xpInfoSource` for this purpose, and all the real information source objects are derived from it. The external values are asked from some external information sources (like measured and model spectrum, external database, user, etc.), which are interchangeable, provided that they can answer the question. The questions can be about the spectrum, components, sample, spectrometer, measurement conditions, etc.

Certain spectrum data formats (see for example [8]) can contain information on whether the energy points are given on the kinetic or binding energy scale. In such cases the spectrum object (see next section) can provide this kind of information; in other cases the user (also a source of information!) shall be asked. For example, the `xpSpectrumBase` object (which is able to tell how the spectrum data point energy data are interpreted) can be used as shown in Appendix A.2.3.

3.3. Using data-evaluation information

Since the goal of the package is to respond to requests from the data-evaluation and data-acquisition software, the method of interaction with them shall also be elaborated. Here the method used specifically to communicate with the wxEWA spectrum

evaluation software [9] is presented. However, it is not too much difference whether the active peak object replies (i.e. an Object Oriented Programming (OOP) implementation is used) to a question about the peak energy or the data-evaluation software reads out some value of the corresponding array and assembles the reply the rule needs.

```
IsEnergyKnownBE(Spectrum) =
{ IsEnergyBE(Spectrum)
  || IsEnergyKE(Spectrum)
    &&
    IsXEnergyKnown(Spectrum)
}
```

It is possible to define a consequent and fully OOP-based model for spectrum evaluation, see [10]. In this scheme, the central object is the "computed spectrum" (see Fig. 2), which contains a list of components (background and peaks) and references to objects "measured spectrum", "sample", "spectrometer", etc. The figure is actually an (incomplete) object interworking diagram for the wxEWA program, but it also illustrates the advantages of the object-oriented design.

One has to teach *only* the "Spectrum component" object to answer the question "Are your energy-related parameters given on the binding energy scale?" and all peaks and backgrounds will inherit this ability. It is enough to implement the "GetPeakEnergy" method in "peak component" and all derived peaks will do it in the same way. This feature can be advantageously used when it is asked if there is a peak with a given energy in the spectrum. The "Computed spectrum" object using its "HasPeakInRegion(EnergyLow, EnergyHigh)" method can ask all "Peak objects", stored in its internal list, and they can reply with a result using the "IsAt(energy)" method. These methods can even actually be invoked in a rule, so that part of the platform-specific rules can be built into spectrum evaluation objects.

3.4. Some general rules

For completeness, the already mentioned rule ("is the binding energy available") given as an example is presented here using the meta-language syntax. Here and below in the rule names, "**BE**" stands for "Binding energy" and similarly "**KE**" for kinetic energy. Also, since these rules involving these types of energy are completely analogous, in the followings only the "**BE**" version is listed, but also the "**KE**" version exists in the implementation.

```
IsEnergyKnownKE(Spectrum) =
{ IsEnergyKE(Spectrum)
  || IsEnergyBE(Spectrum)
    &&
    IsXEnergyKnown(Spectrum)
}
```

A helpful rule could be whether the energy region in question is measured completely:

```
IsRegionMeasuredBE(Spectrum, BE1,
  BE2) =
{  IsEnergyKnownBE(Spectrum)
    &&
    IsInRangeBE(Spectrum.Measured
      LowBE, Spectrum.MeasuredHighBE,
      BE1)
    &&
    IsInRangeBE(Spectrum.Measured
      LowBE, Spectrum.MeasuredHighBE,
      BE2)
}
```

(to decide if a peak energy falls in a certain range, one has to make the evaluation operations in a somewhat wider range, for example, to determine the background).

Also a frequently used rule is whether an energy value falls in some range:

```
IsInRangeBE(BE1, BE2, EnergyBE) =
{  EnergyBE > BE1
    &&
    EnergyBE < BE2
}
```

Since the peaks are objects and they can tell their energy, the rule if a peak has its energy in some range has the form:

```
IsInRangeBE(Peak, BE1, BE2) =
{  Peak.EnergyBE > BE1
    &&
    Peak.EnergyBE < BE2
}
```

In the spectrum there is generally more than one peak, so a rule advising if there is a peak in a given energy range of the spectrum is also useful:

```
HasPeakInRangeBE(Spectrum, BE1, BE2) =
{  IsInRangeBE(BE1, BE2,
   Spectrum.Peak1)     // For all peaks
     ||
     IsInRangeBE(BE1, BE2,
       Spectrum.PeakN)
}
```

4. Expert rules: the ‘Carbon contamination’ rule set

As shown in Section 2, the presented ‘inference engine’ is able to provide the functionality required in everyday data-evaluation practice. As an example for practical applicability, this section describes how the “rule set for carbon contamination” (see [1]) can be implemented using the proposed base package and the domain-specific rules set.

4.1. Is carbon 1s peak present?

The criterion if the C 1s peak is present in the spectrum at all, is

```
HasCarbon1sPeak (Spectrum) =
{  IsRegionMeasuredBE(Spectrum,
   282-2,     462+2)     //Region measured,
   BE available
     &&
     IsRegionMeasuredKE(Spectrum,
       269-2,  275+2)  //Region measured,
       KE available
     &&
     HasPeakInRangeBE(Spectrum,     282,
       288) //C 1s present
     &&
     HasPeakInRangeKE(Spectrum,     269,
       275) //KLL Auger present
     &&
     ! HasPeakInRangeBE(Spectrum, 458,
       462) //Ruthenium absent
}
```

In the rule above some energy values are represented in form of “$x+/-y$”, where “x” is one of the limiting values of the range a characteristic peak shall be found in, and “y” is the amount of extension necessary for evaluating the peak in that energy range safely, see Section 3.4. In the rule above 2 eV value is used. Note that here it seems unnecessary to verify if the region containing the C 1s peak is measured at all and if the given energy values are accessible. And really, they could be verified in the lower-level rules, too. In the latter case, however, they will be executed more than once, so that the performance decreases and the length of reasoning string increases. The inclusion or exclusion of such questions is a matter of strategy and should be debated.

4.2. Is the surface carbon contaminated?

In case C 1s is present in the measured spectrum, one has to verify if the sample contains carbon. So the rule for determining if the sample is carbon contaminated, might be

```
IsCarbonContaminated(Spectrum) =
{  (NOT IsCarbonInSample(Spectrum)
    AND
    HasCarbon1sPeak(Spectrum)
    )
    OR
    HasCarbonContamination(Spectrum)
}
```

The information whether the sample contains carbon can be taken from the sample information for a particular spectrum:

```
IsCarbonInSample(Spectrum) =
{  IsCarbonContained
  (Spectrum.Sample)
}
```

The second half of the rule is a composite rule. Neither of the sub-rules alone is decisive, but they all increase by some amount the chance that the sample is carbon contaminated. The sum then can be below a lower or above an upper threshold, or even between them. The rule then “fires” correspondingly.

```
HasCarbonContamination(Spectrum) =
{  x = 0
```

```
    if IsEnergySeparationInRange
      (ESLow, ESHigh) x = x + 15
    if HasCarbon1sPeak(Spectrum) x = x
      + 20
    if IsShirleyParameterAbove
      Threshold(Spectrum.PeakC1s)  x
      = x + 25
    if IsCarbonSlopeGreater
      (Spectrum) x = x + 25
    ...
    False if x < 20
    True if x > 70
    else Unknown
}
```

4.3. Need for supporting evaluation procedures

Unfortunately, the embedded character rises new problems. Namely, one needs development of new automatic data-evaluation procedures. The two yet unknown comprised rules in rule HasCarbonContamination(Spectrum) above, can be easily expanded:

```
ShirleyParameterAbove
  (Spectrum.Peak1s, ShirleyThreshold)
  =
{  GetShirleyParameter
  (Spectrum.Peak1s)
  > ShirleyThreshold
}

CarbonSlopeGreater(Spectrum) =
{  GetSlopeValue(Spectrum.Peak1)   >
  GetSlopeValue(Spectrum.Peak1s)

    &&
    ...
    GetSlopeValue(Spectrum.PeakN)  >
      GetSlopeValue(Spectrum.Peak1s)
}
```

These new rules have a common characteristic: they need a data-evaluation parameter, belonging to the individual (photo)peaks, that can be evaluated in a non-interactive way. Most data-evaluation methods use the Shirley background as an integral prodecure (valid for the whole spectrum, rather than characteristic of individual peaks) method. Although there exists an interpretation [11] that allows attaching the Shirley-contribution as a tail to the peaks, its use is not typical in widely used data-evaluation software.

A similar situation exists with the 'SlopeValue' parameter [12]. This again, needs to be defined for each peak, and its use limited to the vicinity of the peak. In addition, the slope parameter can be highly correlated with the slopes of other nearby peaks.

Today, the Tougaard-type background evaluation [13] is the physically most correct method for background subtraction. Unfortunately, it is an integral method and is not able to deliver per peak information, like post-peak slope or Shirley height, needed in the "Carbon contamination" rules. Since there exists a possibility [14] to derive a peak tail, equivalent to the Tougaard background, there is hope that similar information from this physically correct model can be derived in the future, too. Otherwise, one would have the options of either to use a "correct" method in the data-evaluation process (with no chance to use the expert system) or to use an "incorrect" data-evaluation method with the expert system.

The lack of any human intelligence in the automated process also raises problems. For example, to decide which peak is the C 1s peak, one needs additional support. The rule 'HasCarbon1sPeak' only declares its existence, but does not give support for the case if more than one peak is present in the energy range, characteristic of the C 1s peak; similarly problematic is when an energy correction is necessary for charging.

In general, the present approach needs well-established, stable, consistent, automated data-evaluation methods. Either these methods shall be given on an algorithmic level (i.e. the missing methods shall be elaborated) or some other methods shall be used instead in the rules. The methods in their present form assume the assistance of an expert person, who will not be generally available in an automated, "embedded" expert system any more.

It should be noted that a different approach to the evaluation of acquired data will be needed for different types of problems. For example, several ranges and/or several energy regions taken at different angles may need to be analyzed simultaneously. Since different rules might be based on different data-evaluation procedures, multi-method data-evaluation programs have to be developed.

5. Using the rules

The described base system provides a facility to build an XPS/AES specific expert system, but the task to provide domain-specific rules remains for the user community. To develop such a community-verified rule set, some tools are necessary. Since the success/failure of a rule depends both on the correctness of the rule and the parameters used (which are provided by the data-acquisition/evaluation software, database handler, etc.) it is a good idea to separate a possible data mistake from a mistake in the rule's logic.

One possible way to reach this goal is provided in the sample program with the package [3]. In this sample, the rules take their input from the elements (checkboxes, text fields, etc.) of a graphical user interface. This method allows one to test the 'net' rules, because these input parameters are completely separated from the rules and are under the user's control. For example, the peak information can be set in a graphic interface page, as shown in Fig. 4. The rules can be verified individually, either as simple or as combined rules. The provided sample application allows one to test the domain-specific general rules (also part of the package), as well as the rules implemented as proposed by Castle and Baker [1]. As a different kind of function of an expert system, it demonstrates how the measurement time can be optimized using the principles suggested by Harrison and Hazell [15].

6. Using wizards

Many experts (see, for example [16]) prefer the wizard style for making logical conclusions and reasoning with the expert system. From the user's point of view, wizards are a safe way to reach some goal, without the need to know many details. From the programmer's point of view, the wizards are specialized and directed dialogs. As they are defined in the multi-platform package [17] wxWindows:

> These dialogs are mostly familiar to Windows users and are nothing else but a sequence of 'pages' each of them displayed inside a dialog which has buttons to pass to the next (and previous) pages. The wizards

Fig. 4. Simulating C 1s peak parameters for the carbon contamination rule.

are typically used to decompose a complex dialog into several simple steps and are mainly useful to the novice users; hence it is important to keep them as simple as possible.

As follows immediately from this definition, using wizards or proceeding without wizards is just a technical question in general and a matter of taste. In the present package (and sample application) both styles can be selected. In addition to using the rules directly as discussed in the previous section, the wizards direct the user to establish the necessary conditions, ask for the requested piece of information, and even force some method of usage via temporary disabling of some operations, making some routes one-way only, etc. For an example see Fig. 5.

The main difference between using wizards and using the rules directly is that the wizard "knows" the necessary conditions as well as the ways they can be established and directs/forces the user to follow the necessary steps. When applying the rules directly, the user has to figure out what is still missing from reasoning he receives from the system. That is, using wizards is a "hard-wired" way of receiving a reply from the inference engine. With a wizard either all phases are passed (during which the needed replies/information pieces are delivered and the logical parameters of the rules established) resulting in some conclusion or the wizard is cancelled, delivering no reply at all. Without a wizard, the user has to know (or can conclude from the reasoning of the conclusion of the unsuccessful trial) which conditions are necessary for the rule to deliver the reply he wants.

In any case, *the logic behind the scenes remains the same*. Although it is probably easier for a beginner to use a wizard, the user will find much slower to reach the goal with wizards than with using the rules directly. Growing familiarity with the rules and conditions will enable a user to proceed at a faster rate.

Also note some important differences. When using wizards, the user is the medium that gives the replies needed by the engine of the expert system. Because

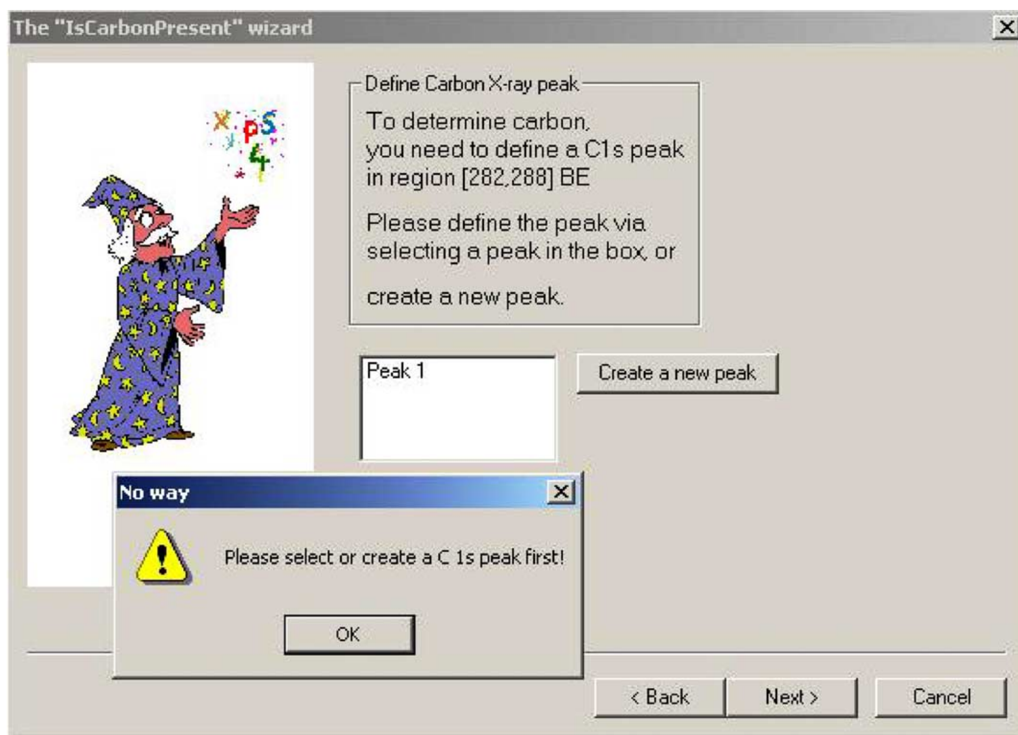

Fig. 5. A wizard page, forcing a user to follow the path.

of this, wizards have some important disadvantages:

- The user's intelligence has to be involved in the decision process.
- The process cannot be automated.
- The user will be the only information source.
- The result will contain some subjective elements (replies).
- Allows 'improvisation'.

On the other hand, wizards have some advantages:

- No direct integration with the acquisition/evaluation software is necessary.
- The user's intelligence can be involved in the decision process.
- Any kind of input information can be used.
- Allows 'improvisation'.

The sample application offers both methods, indicating that *the more important point is the rule and the logical interrelations between various parameters*. The method by which this information is communicated to the inference engine is of secondary importance. The user might reach his goal safely using wizards (and even might learn the 'how's and 'why's), or might directly use the various rules, using the success/failure method.

7. Summary

A new approach for implementing "expert system-like functionality" in data-acquisition and data-evaluation software is proposed. A "reasoning inference engine" is implemented in the described base package, available for free through the Internet [3] for various platforms. The completeness of the package is demonstrated through implementation of the complete 'Carbon contamination' rule set [1]. A simple demonstration program is also available [3], which allows verifying the behavior of the "carbon rule" under different conditions. Using the base package, it is possible to build similar rules and to verify them with applications similar to the demonstration application. The task remaining is *to extract additional rules from existing experience*, to allow their verification by the XPS community and to build them into future acquisition/evaluation software.

Acknowledgements

This work was supported by the project OTKA T046783.

Appendix A

The appendix comprises implementation-related material and is included for the interested reader. It serves mainly as an illustration; the previous material shall be understood without reading it. However, it has close connections to the discussed material and people interested in details of implementation might find it interesting.

A.1. Triple-valued logic 'truth tables'

Possible value range: (Unknown), F(alse) == 0, T(rue) == 1)

(NOT) ! A	A		
	U	F	T
	U	T	F

(AND) A && B		A		
		U	F	T
B	U	U	F	U
	F	F	F	F
	T	U	F	T

(OR) A || B		A		
		U	F	T
B	U	U	U	T
	F	U	F	T
	T	T	T	T

(EQUAL) A == B		A		
		U	F	T
B	U	U	U	U
	F	U	T	F
	T	U	F	T

A.2. Examples of rules

The examples below are taken from the actual working program example. The examples are coded in program language C++, because it allows both using objects (for the rules) and "operator overloading", which

allows using an elegant syntax, quite similar to a natural language. For building a Graphic User Interface for the program, the wxWindows multiplatform package is used [17]. Implementation (maybe in less elegant form) is also possible in other program languages. For better readability, the implementation-specific types, headings, comments, documentation, irrelevant parts, etc. have been removed.

A.2.1. 'Generic' rule

The generic rule contains the following data:

```
HistoryString;   // here is the
                   execution history
                   string stored
Name;            // the name of the rule
InfoSource;      // pointer to the
                   info source
value;           // rules saved value
```

And methods

```
// constructors & destructor
AddStringToHistory :  //contribute to
                        history string
GetValue :            //return the
                        stored logical
                        value
CalculateValue :      //calculate the
                        logical value
                        anew using
                        ''Fire''
Fire :                //return the
                        resulting rule
GetHistoryString :    //return the
                        history while
                        calculating
                        the value
GetName :             //return the
                        name of
                        the rule
GetReasoning :        //return the
                        rule's
                        history in
                        reasoning form
GetValueString :      //return the
                        rule' s value
                        in string form
SetShortcutMode :     //set 'shortcut'
                        or 'complete'
                        evaluation mode
=                     //(assign) operator
!                     //(negation)
                        operator
&&                    //(logical AND)
                        operator
||                    //(logical OR)
                        operator
```

The operators work according to the truth table in Appendix A.1.

A.2.2. The 'Is the binding energy known' rule

Actually, the rule used as an example in Section 3.1 can be programmed as simply as this

```
class xrIsEnergyKnownBE :
  public xpRuleXPS
{ public:
    xrIsEnergyKnownBE( xpInfoSource
      *MyInformator=NULL)
    :xpRuleXPS(MyInformator)
        {Name = "Binding energy known";
          }
    xpRuleGeneric  Fire(void)
        { return xrIsEnergyBE
          (InfoSource)
            OR
            (xrIsEnergyKE
              (InfoSource)
                AND
                xrIsXEnergyKnown
                  (InfoSource)
            );
        }
};// of xrIsEnergyKnownBE
```

The code above is the real, working code for that rule! Of course, the called rules shall be programmed separately, but simply in a similar fashion.

A.2.3. The 'Are the energy data given on kinetic energy scale' rule

This rule is an example for the case when some external information source is used to assemble the value. In this particular case the "spectrum" is directly asked, but for example a "user" object can also

be asked, supposing that he can answer the question 'IsEnergyKE'.

```
class xrIsEnergyKE : public xpRuleXPS
{  public:
    xrIsEnergyKE(  xpInfoSource *MyInformator=NULL)
    :xpRuleXPS(MyInformator)
    {  Name = "Energy-Data are KE";}
    xpRuleGeneric  Fire(void)
    {  xpSpectrumBase *xpS = (xpSpectrumBase *) InfoSource;
        if(xpS)
        {  if (xboolean::True == xpS->GetEnergyBE())
                value = xboolean::False;
              else if(xboolean::False == xpS->GetEnergyBE())
                  value = xboolean::True;
                else value = Unknown;
        }
        else
            value = Unknown;
        if(xboolean::True == value)
            HistoryString << "Energy data are on kinetic scale";
        else if (xboolean::False == value)
            HistoryString << "Energy data are NOT on kinetic scale";
            else HistoryString      <<
              "No spectrum or energy not known";
            return *this;
    }
};// of xrIsEnergyKE
```

A.3. A sample 'Spectrum" external object

```
class xpSpectrumBase : public xpSpectrumObjectGeneric
{  protected:
    wxList  PeakList;  //Peaks attached to the spectrum
    float EnergySeparation;  //
    public:
        xpSpectrumBase(void)
        {  };
        xpSpectrumBase(const xpSpectrumBase& S)
        {    EnergySeparation = S.EnergySeparation;}
        wxList* GetPeaks(void)
            {    return &PeakList;}
        float GetEnergySeparation(void)
        {    return EnergySeparation;}
        void SetEnergySeparation(float ES)
        {    EnergySeparation = ES;}
};//xpSpectrumBase
```

References

[1] J.E. Castle, M.A. Baker, J. Electr. Spectrosc. Rel. Phenom. 105 (1999) 245.
[2] http://www.vide.org/xps.htm
[3] http://xps4xps.sourceforge.net/
[4] Seatle Robotics Society, Fuzzy Logic Tutorial, http://www.seattlerobotics.org/encoder/mar98/fuz/flindex.html
[5] M. Fisch, T. Atwell, Trans. Charles S. Peirce Soc. 11 (1966) 71, or http://plato.stanford.edu/entries/peirce-logic/.
[6] C.J. Date, Database Programm. Design 2 (1989) 50, or http://www.firstsql.com/inulls.htm
[7] http://www.mindview.net/Books
[8] W.A. Dench, L.B. Hazell, M.P. Seah, Surf. Interf. Anal. 13 (1988) 63.
[9] http://wxewa.sourceforge.net/
[10] Comput. Phys. Commun., in press.
[11] J. Végh, J. Electr. Spectr. Rel. Phenom. 46 (1988) 411.
[12] J.E. Castle, I. Abu-Talib, S.A. Richardson, in: Proceedings of the Material Research Society Symposium, vol. 48, Pittsburgh, USA, 1985, pp. 471–479.
[13] S. Tougaard, Phys. Rev. B34 (1986) 6779.
[14] J. Végh, Surf. Interf. Anal. 18 (1992) 545–550.
[15] K. Harrison, L.B. Hazell, Surf. Interf. Anal. 18 (1992) 368.
[16] J.E. Castle, Surf. Interf. Anal. 33 (2002) 196–202.
[17] http://www.wxwindows.org/