Pro/3 – A Production System-type Expert System Shell

Jens Hintze Holm

Note that a few corrections including the PROLOG-representation of sentence rule 7 have been incorporated after the paper was presented in April 2006.

ABSTRACT

Pro/3 is an expert system shell with a knowledge model which can include high volumes of facts, crisp logic rules (PROLOG-type inference) with substantial add-on's for handling statistical and other operations on *sets* of facts, as well as inexact reasoning rules with certainty factors, probabilities and fuzzy sets. Pro/3 can be operated as a backward chaining system, where all queries are resolved by interpreting all facts and rules in the knowledge base on the fly. This is however all but impractical in knowledge bases of some size, and a forward chaining super-structure, which derives sentences once and for all and stores them in the knowledge base, is used instead. The system is then a *production system* where the knowledge base serves as the *working memory*. Possible applications include problems involving inexact classifications (stock selection, market state assessment, medical diagnostics, network diagnostics), event forecasting with certainty rules and rule-intensive applications in general.

I. INTRODUCTION

Pro/3 is an expert system shell, i.e. a system for building applications which can perform expert-like reasoning in a problem domain. This article reviews the system's knowledge representation concepts and the principles of its inference engine. The examples come from a stock picking model which ranks 70 stocks traded at the Oslo Stock Exchange (OSE)

II. FACTS

A **fact** in Pro/3 is called a *sentence*, and consistent with that, *facts* can be expressed as natural language sentences. Two types of sentences correspond to Pro/3 facts. These are (i) sentences with a subject and a predicate, and (ii) sentences with a subject, an object and a predicate.

The sentence structure is based on the entity-relationship data model, such that subjects and objects belong to defined **entity types**, while predicates belong to defined **predicate types** (*unary* predicate types are used in *subject-predicate* sentences, while *binary* are used in *subject-predicate-object* sentences). Entity types consist of a fixed number of **data element types** (predicate types may or may not have data element types). Data element types belong to a set of predefined domains, which include four basic domains: *integer, number, string* and *identifier*, a number of sub-domains including *xml date, xml time, datetime, duration, certainty factor, probability, increment factor, decrement factor* and *membership grade*, and finally a complementary set of *list*-domains. Pro/3 thus has a three-level symbol-structure: sentences, entity- and predicate types and data elements.

Example:

> the financial report with ticker "ODF", operations result 26.0, net income 18.0, current assets 325, total assets 1972, current liabilities 135, equity 702, currency US Dollar and sequence no 10 is published for the accounting period with year 2005 and period 2 with publication day "2005-08-24"![1]

Sentence 1 is a subject-predicate-object type sentence, where the subject is a *financial report*-entity and the object is an *accounting period*-entity. The *financial report* entity type has nine data element types (*ticker, operations results, net income* etc), the *accounting period* entity type has two data elements, while the *is published for*-predicate type has one data element (*publication day*). (Note that all

natural language representations of sentences, sentence rules and function definitions are terminated by **!**).

Sentence 1 corresponds to the PROLOG-fact:

```
pIsPubFor(eFinRep("ODF",26,18,325,1972,135,70
2,iUSD,10),eAccPer(2005,2),"2005-08-24")
```

represented as a **term** in the knowledge base:

```
cmp(":-", cmp("pIsPubFor", cmp("eFinRep",
str("ODF",int(26),…,int(10)]), cmp("eAccPer",
int(2005),int(2)], str("2005-08-24")],atom("true")])
```

The combination of predicate type and entity type(s) in a sentence is referred to as the sentence's *type*, such that sentence 1 belongs to the *financial report is published for accounting period* **sentence type**.

Translation between natural language-format knowledge and internal format knowledge, is facilitated by a **terminology** and a **sentence model**. The terminology is a set of terminological *facts*, which state e.g. that *US Dollar* is an *identifier* (represented internally as *iUSD*), rather than say two variables *US* and *Dollar* (words starting with uppercase letters generally represent variables in Pro/3). The sentence model is a set of facts which define, among other elements, the predicate types, entity types and data element types.

All knowledge declared to Pro/3, i.e. rules and facts (including terminology, sentence model and other meta-knowledge facts), are stored in a database referred to as the **knowledge base**. The knowledge base is implemented either as one or more Visual Prolog *chainDBs* or as an SQL-database, which Pro/3 accesses via an ODBC interface.

III. QUERIES

There are two classes of queries in Pro/3, that is, *queries which unify with a set of sentences* and *queries which unify with a single value*. A **sentence query** has the same structure as a *sentence* with part(s) of the sentence made variable - either implicitly, explicitly by replacing part(s) of the sentence with

variable(s), or by giving conditions to one or more of the data element types in the sentence.

Examples:

```
which financial reports with net income greater than
10.0 are published for what accounting periods with
year 2004?2
X is published for Y for publication day greater than
"2005-02-16"?3
```

2 translates into the PROLOG-query

```
pIsPubFor(eFinRep(_,_,X1,_,_,_,_,_,_),eAccPer(X2
,_),_), X1>10.0, X2=2004
```

which trivially can be resolved by a PROLOG-type inference engine. The query expression will be unified with the *financial reports are published for accounting period*-sentences (facts) satisfying the given *net income*- and year-conditions. Note that interrogatives (*which*, *what* and, *who(m)*) indicate that the succeeding entity type is variable. This is basically syntactic sugar since both *which* and *what* can be dropped from 2 without changing the meaning. 3 which uses variables explicitly (*X* and *Y*), has a different interpretation, since any entities can unify with X and Y (not only *financial reports* and *accounting periods*).

Single-value queries return (unify with) a *single value* (or a *series* of single values if *non-deterministic*), such as the value of a given data element in a sentence. Other varieties return the number of sentences satisfying a given condition or the value returned by a *function* call (section V *under*). Single-value queries are important because *query-type inexact rules* include one such query (section VII *under*) - the rule uses the answer directly (or via a mapping) as its return-value.

Examples:

```
what equals the value of net income for financial
report with ticker "ODF" which is published for
accounting period with year 2005 and period 2?4
what = faculty given integer 5?5
what equals today? 6
```

4 can be resolved using the same inference logic as is used for the sentence query in the examples over. 5 is a call to a user-defined *function*, which also is resolved using normal PROLOG-type inference (= and *equals* are synonyms). 6 is a call to the *today*-function, which is built into the inference engine.

IV. SIMPLE SENTENCE RULES

Sentence rules define how sentences can be derived from other sentences.

Stocks at OSE are priced in NOK, while financial reports in some cases are given in other currencies (sentence 1 has US Dollar-denominated figures). To compute price earnings and other indicators, and to facilitate comparisons between figures in different currencies, it is necessary to convert all figures to NOK. Applicable conversion rates are given by a set of facts

```
exchange rate with rate=6.52 and currency=US
Dollar is quoted for trading day "2005-08-24"!
```

NOK-equivalent financial reports can be expressed in a new sentence-type *nok-equivalent financial report is published for accounting period*. This sentence type is derived by the following rule:

```
if the financial report with ticker T, operations result
OR, net income NI, current assets CA, total assets
TA, current liabilities L, equity E, currency C and
sequence no N is published for accounting period
with year Y and period P with publication day D and
the exchange rate with rate R and currency C is
quoted for trading day D, then the nok-equivalent
financial report with ticker T, operations result
(OR*R), net income (NI*R), current assets (CA*R),
total assets (TA*R), current liabilities (L*R), equity
(E*R) and sequence no N is published for
accounting period with year Y and period P with
publication day  D!7
```

Note that words starting with a capital letter represent *variables* (unless explicitly declared as identifiers). The rule is translated into a term via the following PROLOG-style representation:

```
pIsPubFor(eNfinRep(T,OR,NI,CA,TA,L,E,C,N),eAcc
Per(Y,P),D):-
```

```
pIsPubFor(eFinRep(T2,OR2,NI2,CA2,TA2,L2,E2,N
2),eAccPer(Y2,P2),D2),
pIsQuoted(eExRate((R3,C3),D3),
T=T2, OR=(OR2*R3), NI=(NI2*R3), CA=(CA2*R3),
TA=(TA2*R3), L=(L2*R3),E=(E2*R3),C=C2,N=N2,
Y2=Y, P2=P, D=D2, D3=D2.
```

Natural language text format is generally ***not*** practical when writing rules, as will be seen from 7. The rule is difficult to read and prone to errors e.g. by mistakenly omitting data elements, by misspellings or otherwise. The need for many user-defined variables is also problematic. A better approach is to use Pro/3's **graphical rule editor**, by which rules are drawn as *trees*, and rule constituents such as sentence-types, data element types and operators, are picked from a palette. A *rule-tree* corresponding to 7 is shown in Figure 1.

V. FUNCTIONS

Functions are mainly used to keep computational logic out of the sentence rules, and thus make rules simpler with less redundancy (where the same computational logic would have had to be repeated in different rules).

For reasons which will become apparent in section VI, accounting periods (given by a year and a quarter), need to be related to actual dates. The simple rule in Figure 2 is introduced for this purpose. The rule uses the two functions *accounting period start day* and *accounting period end day*.

A function consists of a declaration (its visible part which is considered part of the *sentence model*), and one or more definitions (its internal part or *clause(s)*).

Example

Declaration:
```
the deterministic function with NL-name accounting
period end day, domain name XML_DATE and data
element list year and period is in the sentence
model with segment name "OSE"!
```

Definitions:
```
accounting period end day is defined by:
this period = 4;
```

```
Date = xml date string given year this year, month
number 12 and day number 31;
return value = Date! 9
```

```
accounting period end day is defined by:
this period <> 4;
NextPeriod = (this period + 1);
NextDay = accounting period start day given year
this year and period NextPeriod;
Date = (NextDay - "P1D");
return value = Date 10
```

The *accounting period end day*-function has
two definitions (clauses) each framed by an *is
defined by* statement and a *return value*
statement (terminated by **!** – it is not a *cut*).
Statements are otherwise terminated by **;**. The
second-last statement in 9 is a call to another
function *xml date string*, which converts three
integers into a date (in XML string format).
The second-last statement in 10 subtracts a
one-day long *period* (XML-format *period*
literal) from a *date-variable* and assigns
(*binds*) the result to the variable *Date*. **this** is a
qualifier for *data element types* which
represent the formal parameters inside the
function (the same data element types (names)
could play other roles, and thus create
ambiguities if not qualified).

The PROLOG-format definition (*under*)
corresponding to 9, shows that functions can
be interpreted directly by a PROLOG-type
inference engine:

```
fAccPerE(FAccPerE,FAccPerE__aYear,FAccPerE_
_aPeriod) :-
FAccPerE__aPeriod = 4,
p3_fDtXml(Date,FAccPerE__aYear,12,31),
FAccPerE = Date.
```

The terminology has assigned the internal
name fAccPerE to the *accounting period end
day*-function, while its two parameter data
element types are known as *aYear* and
aPeriod respectively.

VI. SET-TYPE SENTENCE RULES

The rule in Figure 1 is defining the conversion
of *financial* report-figures to NOK, based on
the rate prevailing on the publication date.
This date is often more than a month after the
end of the accounting period in question, and
it makes better sense to use the *average* rate

for the period (quarter) instead. A rule which
concludes a sentence-type like *accounting
period has average exchange rate* is then
needed.

Set rules conclude sentences derived from *sets*
of other sentences. For the problem at hand,
one *accounting period has average exchange
rate*-sentence needs to be concluded for each
set of the 65 or so *exchange rate is quoted*-
sentences pertaining to the same accounting
period and currency. The rule which specifies
this is shown in Figure 3.

Set rules associate one of the predefined Pro/3
procedures with at least one of the data
element types in the *concluded* sentence-type.
This looks superficially like a function call,
but has a different interpretation. In the rule in
Figure 3, data element *rate* is bound to the
AVERAGE_OF procedure. This procedure has
one number-domain formal parameter, which
is bound to the *rate*-data element type from
the condition's *exchange rate is quoted*
sentence-type. The AND-ed (joined) set of
accounting period is defined and *exchange
rate is quoted* sentences, is called the
condition set.

Let's review in some detail how the inference
engine resolves the query *which accounting
period has what average exchange rate?*

The query `pHas(eAccPer(Z1,Z2), Z3, Z4)` 11
unifies with the conclusion in Figure 3, and in
the next step, with its condition. However,
unlike a simple rule like 7, the condition of a
set rule cannot be resolved directly by a
PROLOG-type inference engine (although it
technically is a *term* in the knowledge base):

```
p3_iRsta(str1, str2, str3, Var1, Var2, Var3, Var4)
```

The term includes three pieces of information:

- *str1* is a string-representation of a
  query-term which resolves with the
  condition   set:
  ```
  pIsDefined(eAccPer(X1,X2),X3,X4),
  pIsQuot(eExRat(X5,X6),X7)),X7>=X3,X7<
  =X4.
  ```

- *str2* is a string-representation of variables (terms) corresponding to the **concluded data element types**: [Y1,Y2,Y3,Y4].
- *str3* is a string-representation of the **expressions (terms) assigned to the concluded data element** types: [X1,X2,p3_rAvg(X5),X6], where *p3_rAvg* is the internal name of the *AVERAGE_OF* procedure).

The inference engine recognizes the *p3_iRsta* header, unpacks the condition query, and invokes another instance of the engine to determine the actual condition set from this query. This set of sentences, and the list of assignments, is then used to unify the original query variables *Z1-Z4* 11 with values computed from the set (via the *Var1-Var4* variables.

The rule in Figure 3 is an example of a *statistical*–type set rule, which supports the use of 13 different procedures: *average, mean, sum, standard deviation, maximum, minimum, start value, end value, start point, end point, maximum point* and *minimum point*. The last six procedures view the condition set as an *ordered* set (defined by a specified ordering data element type in the condition such as e.g. *trading day*). There is also a *count*-procedure which simply returns the size of the condition set.

A second type of set rules broadly follows the pattern of the statistical set-rules, however the procedure for deriving the concluded sentences is more complicated, and will not be discussed here. Interpretation of the rules involves various manipulations of the condition set (before sentences concluded by the rule are derived from the set). These manipulations include *interpolation/extrapolation* (e.g. to compensate for missing sentences (values) in a time-series type of context); *ranking* which often are used when working with certainty factors and membership grades (section VII); measurement of *correlation* between data element types in the set, besides some *selection* operations.

VII. INEXACT RULES

The OSE stock picking model includes 125 sentence rules, which derive about the same number different sentence types. These sentences include stock pricing indicators, moving averages, volatility and earnings and price growth measures etc, which form a body of *crisp* knowledge, relevant for the goal of picking attractive stocks. The knowledge is crisp because it is derived by well-defined numerical methods, essentially without elements of vagueness or subjectivity.

When proceeding further towards the goal of ranking stocks based on "attractiveness", one will soon realize the absence of a generally accepted, well-defined and objective method. There are several common approaches and various rules-of-thumb, but no numerical method that brings it all together. This is actually fairly obvious, since attractiveness of a stock is a measurement of the likelihood of a host of future events, which mostly are very difficult to predict. A "good" method, however, does only have to produce a ranking slightly better than a random one to be useful. The OSE model uses a combination of rules-of-thumbs represented as **fuzzy sets**.

An essential assumption in traditional "crisp" logic is that *something* (an *element* belonging to a certain *universe*) either is a member of a given *set* or it is *not a member*, in which case it is a member of the *complement set* (in the *universe*). In fuzzy set theory this is not so. Fuzzy set membership is qualified by a **membership grade**, given by the set's *membership function*, which is a mapping from an element (represented by a parameter or parameters in Pro/3), to a real number in the range [0,1]. 0 means that the element is definitely *not* a member of the set, 1 means that the element definitely *is* a member of the set, while values in the range <0,1> mean that the element *is a member to a degree* (the higher the value, the higher the degree of membership). An element can both be a member of a fuzzy set and its complement.

The OSE model includes the fuzzy set of *attractive stocks*, which has two parameters *ticker* (which identifies the stock), and *evaluation day* which specifies the date on which the attractiveness is assessed. A stock might be very attractive at 5 September 2005 (based on what is known at that date), but quite unattractive one month later. The model's capability to assess attractiveness for a series of historical dates, is very useful since it can be used to automatically correlate these assessments with the actual return of holding the stock different periods after the dates (ref. correlation-type set rules in section VI). The measured correlations serve the purpose of model validation and improvement.

Fuzzy sets and certainty rules belong to a category of knowledge in Pro/3 called *inexact rules*. An inexact rule is a function which returns a simple value (typically a *certainty factor* or *membership grade*). The function usually has one or more input parameters. The semantic interpretation of a reference (call) to the function is:

In case of fuzzy sets: the function is a fuzzy set *membership function*, and the returned value is the *membership grade* of the set element identified by the actual parameters

in case of certainty rules: the function and the actual parameter values is a *proposition*, and the returned value is a *certainty factor* i.e. a measurement of the certainty of this proposition

Certainty rules and fuzzy sets correspond to different *sources* of uncertainty. Certainty rules are generally used where the *knowledge* of the problem domain is imprecise, while fuzzy sets are used where the knowledge *classifications* (knowledge concepts) are imprecise.

Inexact rules are mostly about *calling other inexact rules* in various contexts, and combining or otherwise using the values returned from the called rules in different ways. The structure of rule calls is a directed acyclic graph. Figure 4 shows a small sub-set of the *attractive stocks* graph, i.e. the part which defines the *stocks with good result per share growth* fuzzy set. A tree-representation of the graph is shown in Figure 5.

With the sole exception of **data rules**, all inexact rules call at least one other rule. While data rules *can* do one type of rule calls known as *context calls*, they generally do not call other rules, and thus are terminal nodes in the graph. The purpose of a data rule is to return a value on the basis of either **a query to the knowledge base** (single-value query), or by **questioning the user** interactively.

Inexact rules derive their return value in three ways:
- By calling other inexact rules and combining or *mapping* these called rules' return-values in various ways. Fuzzy set operations include *intersection* and *union (standard, algebraic product, bounded difference, drastic), complement* and *ordered weighted average*. Certainty rule operations include *and, or, not, bayesian* and *combination*. There are also four support-type rules: *switch-rules, parameter-rules* and *map-rules*.
- By querying the knowledge base (with or without mapping)
- By questioning the user (with or without mapping)

Inexact rules are entered and maintained in a set of Pro/3 form-windows. They do not have any natural language representation, but can be represented as text using XML-format. Figure 6 shows the *attractive stocks* rule, which is defined as the result of combining five other fuzzy sets (*stocks with good earnings growth, stocks with good result per share growth* etc) through an *ordered weighted averaging* operation.

The *attractive stocks* rule is not called by other inexact rules - it is called by a **sentence rule** which defines *stock has buy rating* sentences as is shown in Figure 7.

Let's review how the inference engine resolves the query *which stocks have buy rating*. The only novelty here is that the *buy rating* data element unifies with a call to the *attractive stocks* inexact rule, which can be evaluated when the *ticker* and *day* parameters have been unified with values from the condition sentences. The *attractive stocks* rule can only be evaluated after all its called rules have been evaluated, recursively, until the terminal nodes (*data rules*) of the inexact rule graph are reached. Data rules are evaluated by processing a single-value query or by questioning the user (interactively). The return-values from the data rules are returned backwards through the graph until eventually a membership grade for the *attractive stocks* fuzzy set is determined.

Pro/3 stores the complete history of inexact rule evaluations in the knowledge bases (as *facts*). This includes the evaluation of each individual rule for all parameter combinations, as well as the evaluation of the rule network itself as a tree-structure corresponding to the rule graph. These facts serve the purpose of answering *why* and *how* a certain conclusion has been reached.

VIII. PRO/3 AS A PRODUCTION SYSTEM

The previous sections have briefly reviewed the four knowledge representation concepts (knowledge categories) in Pro/3 – **sentence types**, **sentence rules, functions** and **inexact rules**. Pro/3 analyzes dependencies between members of these categories, and maintains the dependencies as a directed acyclic graph known as the ***knowledge dependency graph***. The direction of the graph follows the *if-then* relationships of the knowledge. It includes the following nine types of dependencies:

- **sentence rule → sentence type** : sentence rule concludes sentence type
- **sentence type → sentence rule** : sentence type is in sentence rule's condition

- **sentence type → function** : sentence type is referred in function (function definition)
- **sentence type → inexact rule** : sentence type is referred in an inexact data rule's *query*
- **function → function** : function is called by function
- **function → inexact rule** : function is referred/called in inexact rule (data rule)'s *query*
- **function → sentence rule** : function is referred/called in sentence rule
- **inexact rule → inexact rule** : inexact rule is called by inexact rule
- **inexact rule → sentence rule** : inexact rule is referred/called in sentence rule

Sentence types (facts) which are entered into the knowledge base, will form the *start nodes* of the graph (nodes without predecessors), while the sentence types not forming part of any sentence rule condition nor referred in query rules or functions, will form the *terminal nodes* (nodes without successors). These are the ultimate conclusions of the expert system. No other nodes can meaningfully be terminal nodes (sentence rules *must* have both a condition and a conclusion), while otherwise function/inexact-rule terminal nodes would represent function/inexact rules not called by any other rule or function, and thus not playing a meaningful role). Functions not calling any other function and without references to *facts*, and *question-type* inexact rules (without context call), will also form start nodes.

Example:

The subset of the OSE-model dependency graph in Figure 8 has three start nodes (*financial report is published for accounting period*-sentence type, *exchange rate is quoted*-sentence type and the *xml date string*-function. The graph has a single terminal node *nok-equivalent financial report is published for accounting period*-sentence type.

If the knowledge nodes in Figure 8 represented an entire knowledge base, the knowledge base could be used without further ado, even if the published financial reports were in the thousands. Queries like *which nok-equivalent financial reports are published for which accounting periods* could be processed by the inference engine on the fly, i.e. by starting with the conclusion (the answer to the query), and from there chaining backwards through all relevant rules and facts (the normal strategy of a PROLOG-type inference engine).

The OSE-model, however, includes some 380 knowledge nodes and more than 500 dependencies between them. Deriving some of the terminal node sentence types involves derivation of more than a 100,000 sentences in intermediary derivation steps. Backward chaining through such volumes of knowledge every time a query is processed, would be very processing-intensive (un-workable in practice). The normal approach in Pro/3 is rather to operate the system as a *forward chaining production system*, such that all possible sentences are derived once and for all, and then stored in the knowledge base (the production system's *working memory*) as is illustrated in <u>Figure 9</u>.

The forward chaining sentence derivation strategy is based on the knowledge dependency graph, which defines the required sequence of sentence derivations, and also tracks the status of derivations. Two additional mechanisms are employed, firstly that of distinguishing between sentences entered by the user and sentences derived by the system, and secondly that of distinguishing between two *states* of sentence rules (active and inactive). Inactive rules are ignored by the inference engine, which is just what we want when all the sentences which can be derived from the rule have already been derived and stored in the knowledge base. Active rules, on the other hand, are always considered by the inference engine, and that is needed during step 3 in Figure 9, or when the knowledge base is operated by using backward chaining only (i.e. without the forward chaining super-structure). Rules are in this case active all the time, and no derived sentences are stored in the knowledge base.

Unlike *entered* sentences, derived sentences are deleted and re-derived by the system during the sentence derivation process (step 1 and 4 in Figure 9). Re-derivations will only take place when needed, that is, when rules or sentences which determine a derived sentence have been changed. If for example an *exchange rate is quoted*-sentence in Figure 12 has been altered (or the sentence rule in Figure 3 has been changed), then all dependent sentences (*accounting period has average exchange rate* and *nok-equivalent financial report is published for accounting period* sentence types), must be re-derived. Dependent sentences include all successor sentence types (recursively) in the knowledge dependency graph.

IX. OBSERVATIONS
An expert system can be designed to serve the role as an *advisor to a non-expert*, or as an *assistant to an expert*. Pro/3's general-purpose input/output facilities and inference concepts, may generally serve the latter application better. A Pro/3 expert system can reason with volumes of facts that are beyond the capacity of a human expert. Secondly, the modeling of the *problem domain* through the iterative process of formulating, validating and re-formulating the expert system, could give the expert valuable new insight into the rules and relations that govern the domain.

Limited scope for problem-specific optimization is a significant problem with general-purpose knowledge representation systems. Expert systems created with Pro/3 could suffer from serious or even crippling performance issues. This stresses the difficulty of liberating the process of knowledge formulation from considerations of expert system implementation and operation.

Visual Prolog is a premier development tool for large and complex applications. Visual Prolog 6.2 is efficient and reliable, and offers

a rich set of class libraries and powerful development tools including source-level debugger and profiler.

X. ACKNOWLEDGEMENTS AND REFERENCES

1. Refer to **www.rudstrand.com** for comprehensive documentation of Pro/3.
2. Pro/3 is developed in **Visual Prolog** v.6.2 (www.visual-prolog.com), and uses several foundation classes including **chainDB** (permanent and run-time database), **odbc** (alternative interface to SQL databases), slightly modified versions of **vpiEditor** (editor), **vpiTableed** (fact browsing and data entry) and **vpiTree** (knowledge dependency tree drawing). **pie** was the starting point for the inference engine (**PDC Prolog Reference Guide** (1992)), while the inexact rule concept has its origin in the PEXPERT certainty rule-based expert system shell (**PDC PROLOG Professional User's Guide** (1990)).
3. **PROLOG: A Relational Language and its Applications**, John Malpas, Prentice Hall, NJ (1987).
4. **Introduction to Expert Systems**, Peter Jackson, Addison-Wesley (1990).
5. **Artifical Intelligence: Structures and Strategies for Complex Problem Solving**, George F. Luger, Addison Wesley (2003).
6. **Fuzzy Sets and Fuzzy Logic: Theory and Applications**, George J. Klir and Bo Yuan, Prentice Hall, NJ (1995).
7. **PROLOG for Natural language Processing**, Annie Gal, Guy Lapalme, Patrick Saint-Dizier and Harold Somers, John Wiley and Sons (1991).
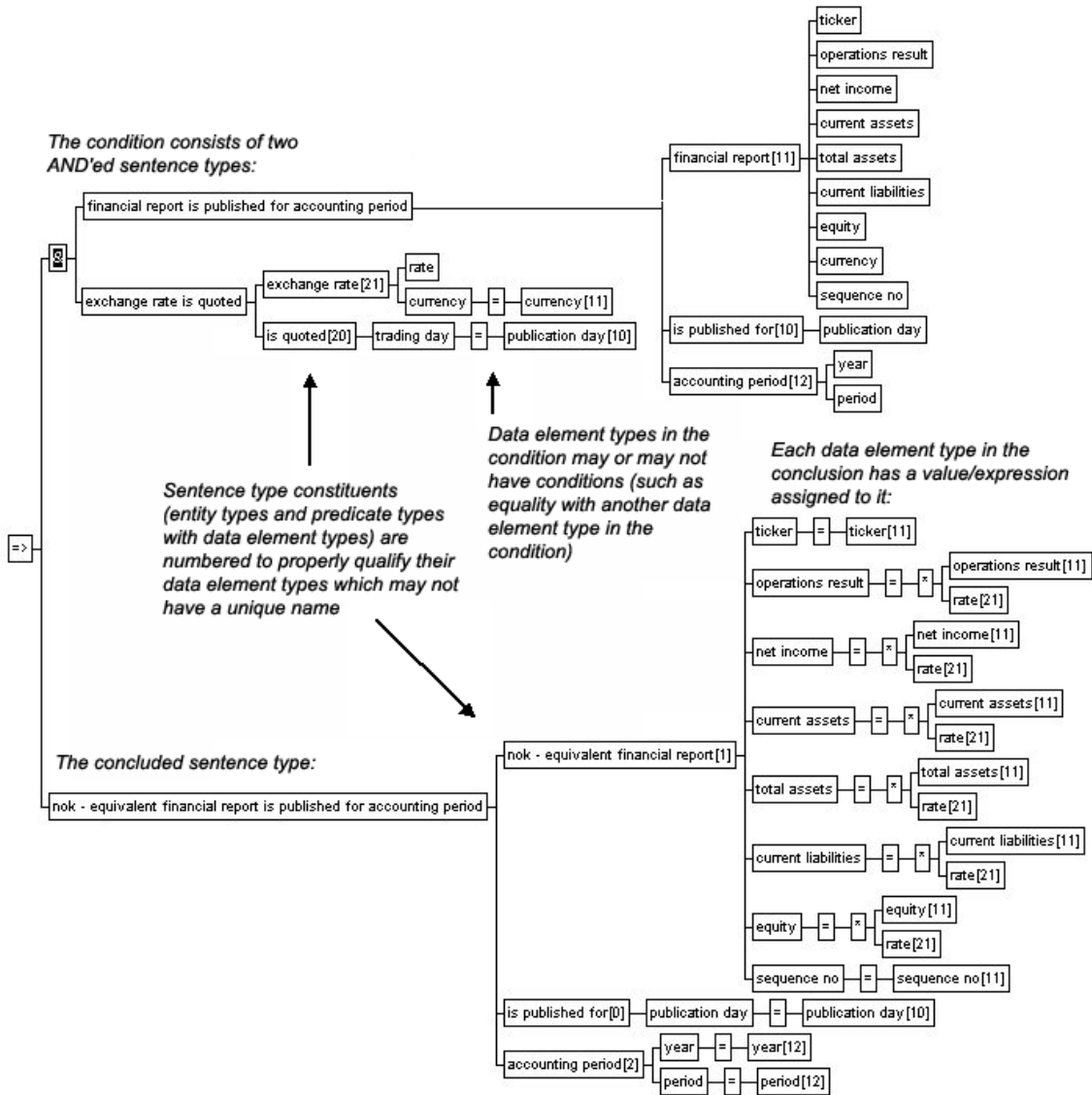
Figure 1 The root-node of the tree encodes the rule-type (in this case a simple implication rule represented by =>). The root-node has two branches. The lower branch represents the **conclusion**, while the upper represents the **condition**. The condition typically branches into several sub-branches combined with logical AND (&) or OR (v). The conclusion is a sentence-type structure (with entity type and predicate type constituents on the second level), and the value expressions assigned to the data element types on the lowest level. It will be noticed that the sentence-type constituents are numbered (e.g. exchange rate[**21**]). This numbering is required to properly qualify the constituents' data element names when these are referred in expressions in other parts of the tree.
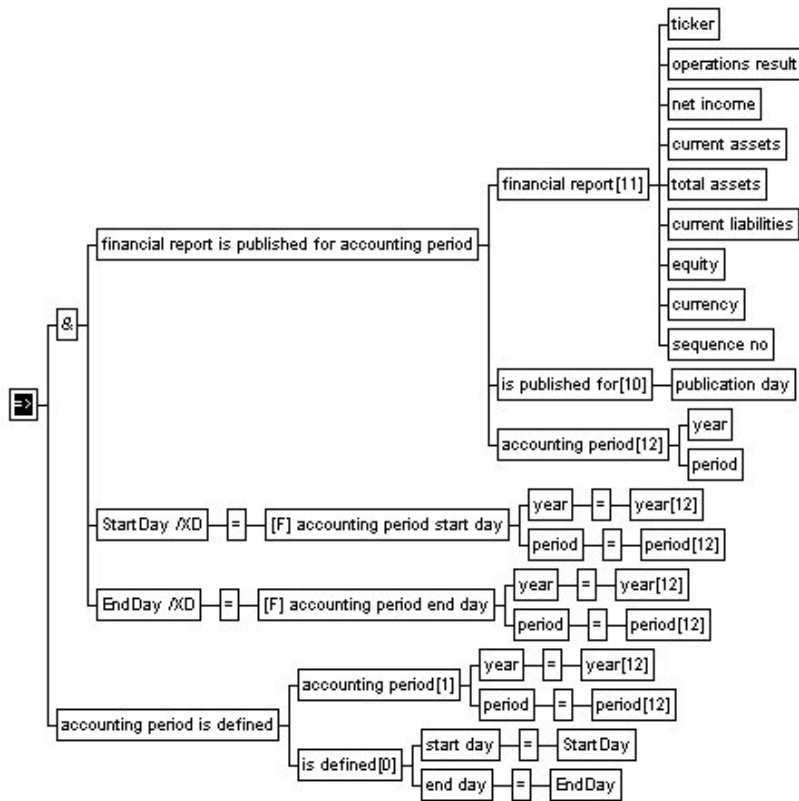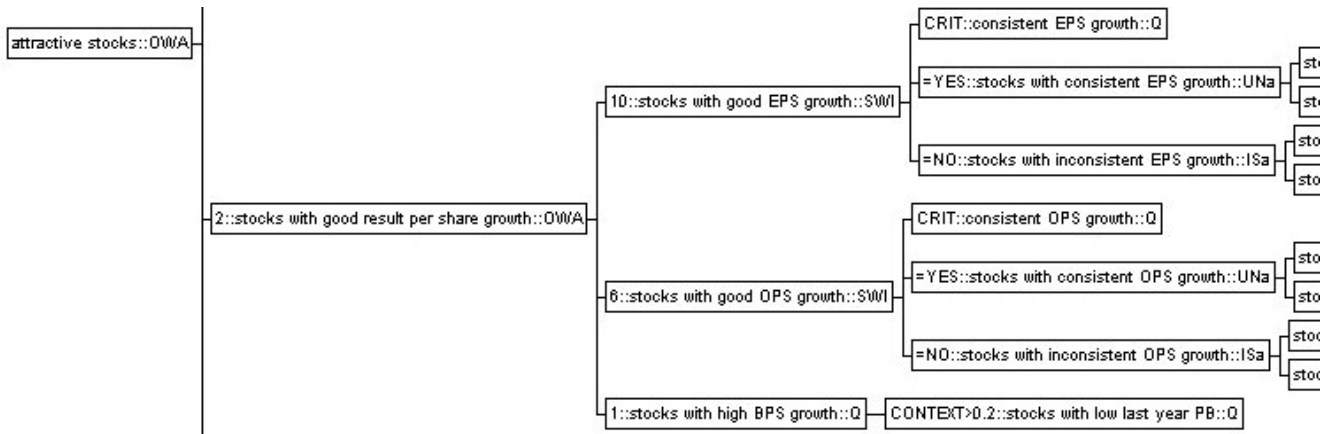
Figure 2 Simple sentence rule which calls the functions *accounting period start day* and *accounting period end day*.
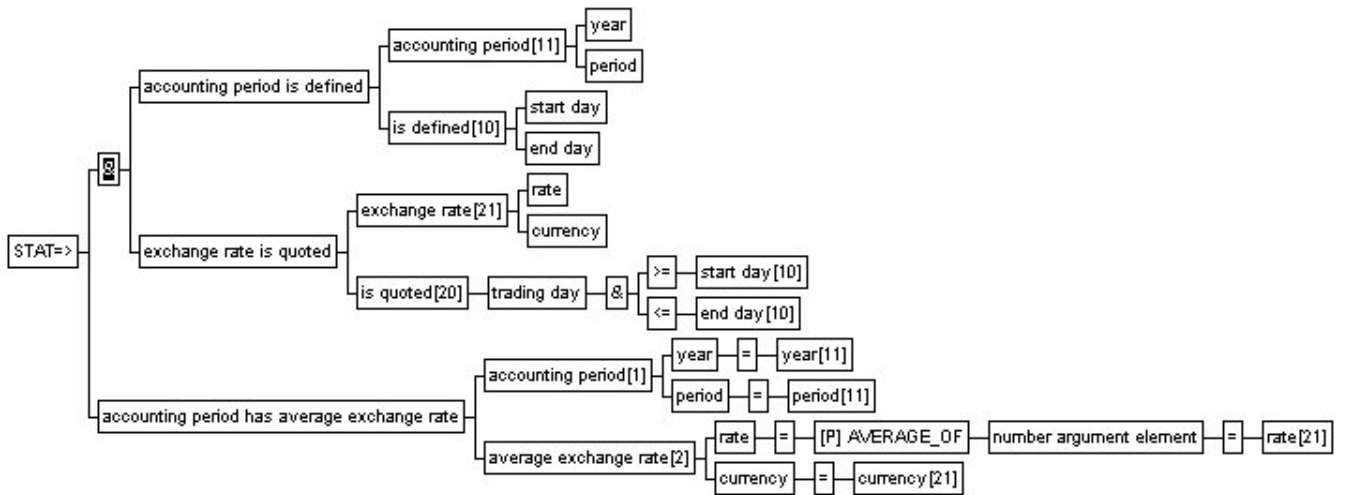
Figure 3 *accounting period has average exchange rate*-set rule.
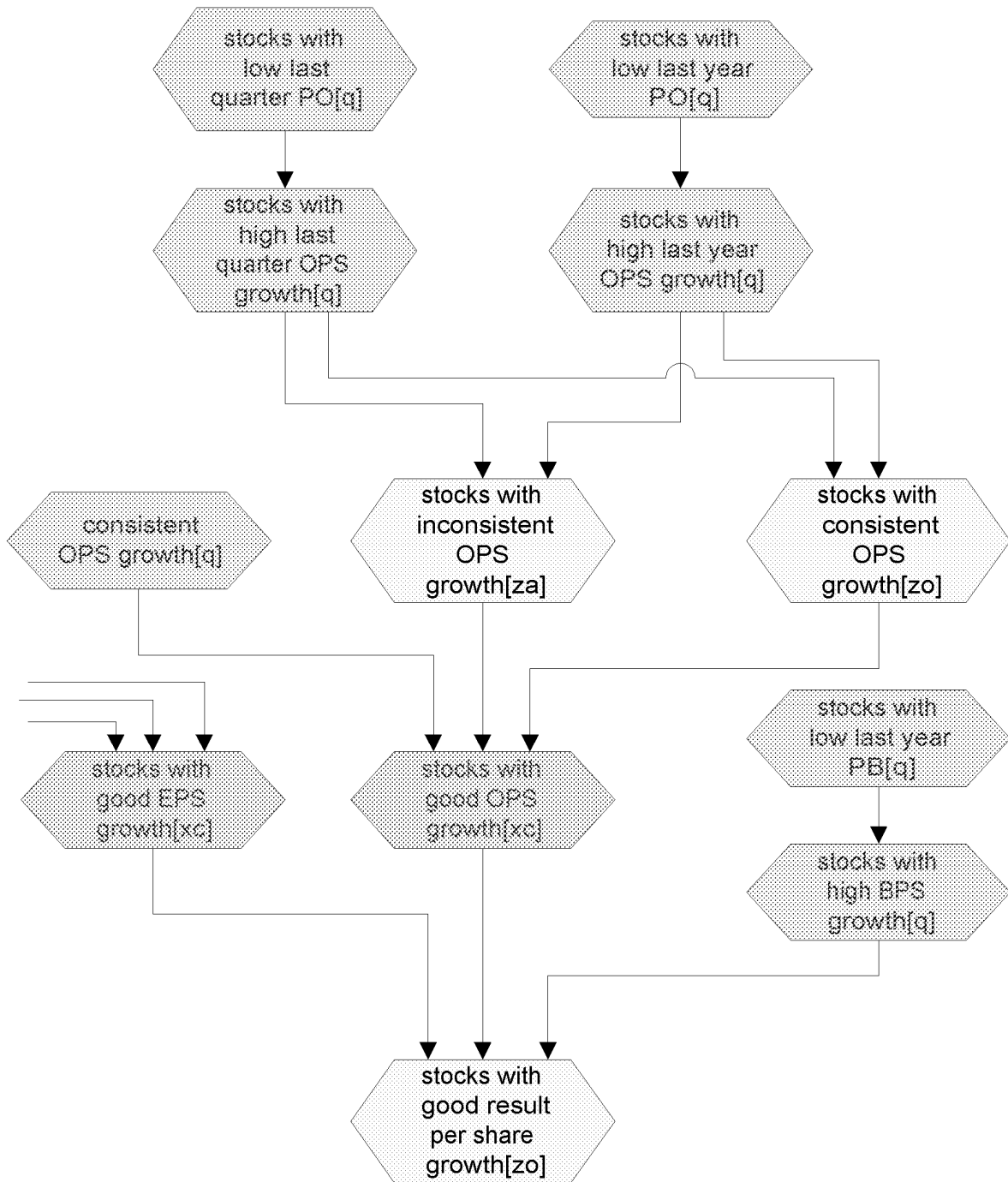
Figure 4 A small sub-set of the *attractive stocks* graph (**zo** denotes an *ordered weighted averaging*-set, **za** an *intersection*-set, **xc** a *switch*-rule, while **q** denotes a query-rule).

Figure 5 Tree-representation of the graph in Figure 4.

```xml
<?xml version="1.0" ?><pro3Knowledge version="4" release="9" build="1"
createdTime="2005-11-13T11:07:14.000" author="Pro/3" xmlns="http://rudstrand.com/xml/" >
<inexactRule format="NL" realm="INEXACT_RULES">
    <fuzzyOWA_Set><set>attractive stocks</set><parameter>ticker</parameter><parameter>day</parameter>
      <OWA><weight>1</weight><set>stocks with good earnings growth</set>
                      <parameter>ticker</parameter><varVal>this ticker</varVal>
                      <parameter>day</parameter><varVal>this day</varVal></OWA>
      <OWA><weight>2</weight><set>stocks with good result per share growth</set>
                      <parameter>ticker</parameter><varVal>this ticker</varVal>
                      <parameter>day</parameter><varVal>this day</varVal></OWA>
      <OWA><weight>10</weight><set>stocks with low price indicators</set>
                      <parameter>ticker</parameter><varVal>this ticker</varVal>
                      <parameter>day</parameter><varVal>this day</varVal></OWA>
      <OWA><weight>1</weight><set>stocks with low risk factors</set>
                      <parameter>ticker</parameter><varVal>this ticker</varVal>
                      <parameter>day</parameter><varVal>this day</varVal></OWA>
      <OWA><weight>10</weight><set>stocks with other price increase indicators</set>
                      <parameter>ticker</parameter><varVal>this ticker</varVal>
                      <parameter>day</parameter><varVal>this day</varVal></OWA>
    </fuzzyOWA_Set></inexactRule></pro3Knowledge>
```
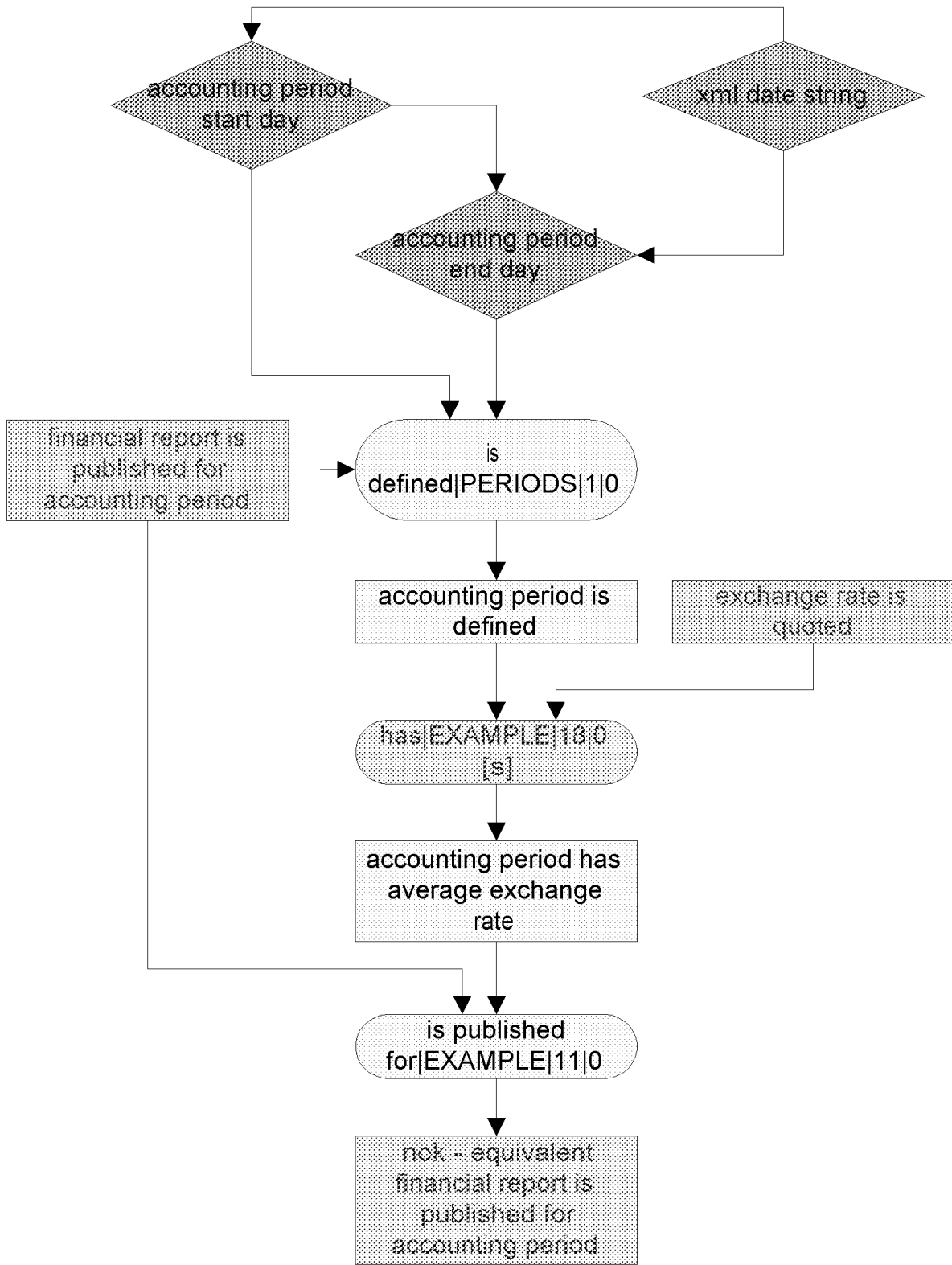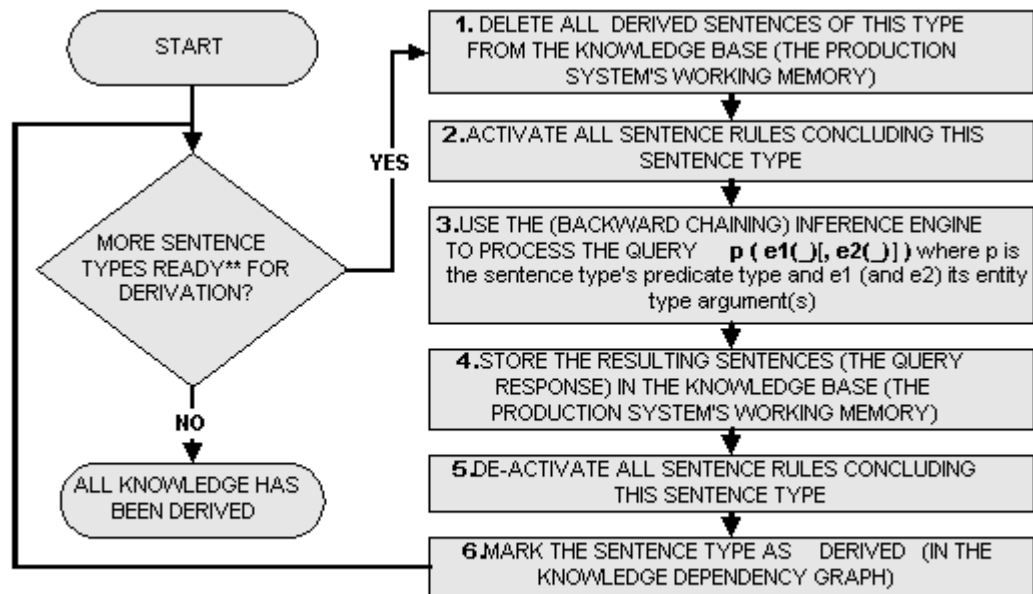
Figure 6 XML-representation of inexact rule (fuzzy set) *attractive stocks.*
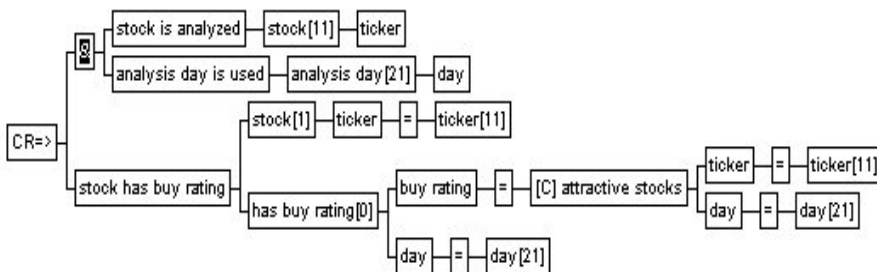
Figure 7 Sentence rule calling the *attractive stock* inexact rule.

Figure 8 Subset of the OSE-model dependency graph. Rectangles represent sentence types, rounded rectangles represent sentence rules, while diamonds represent functions.

Figure 9 Pro/3 forward chaining production system processing model.