LOGIC PROGRAMMING AS A SOFTWARE ENGINEERING TOOL

Gregory Lazarev
and
Winston G. Gresov
GHL Associates
Haverford, Pennsylvania
Logic Programming as a Software Engineering Tool

Gregory Lazarev

and

Winston G. Gresov

GHL Associates

Logic programming is based on the idea that the first-order predicate calculus (in Horn clause form) can be used directly as a programming language. PROLOG (PROgramming in LOGic) is the best known example of this kind of language. Virtually unknown in the United States until a few years ago, PROLOG has now become one of the most popular and promising of artificial intelligence tools. This fact should not be surprising, in view of the very generic nature of logic programming as a tool. Several PROLOG interpreters (and compilers) provide good performance. A number of commercial applications have been written in PROLOG, and more are just a matter of time.

Let's clarify the above statement. Behind it is a quite obvious idea that our knowledge about the world can be represented as facts (e.g. "Mark is male") and if-then rules (e.g. "X is a parent of Y if X is the father or the mother of Y"). The following example, written in PROLOG, illustrates the principle.

```
male(mark).
male(mike).
female(kathy).
father(greg, mark).
father(greg, mike).
mother(susan, mark).
parent(X, Y):- father(X, Y).
parent(X, Y):- mother(X, Y).
brother_of(X, Y):-
    male(X),
parent(Z, X),
parent(Z, Y).
```

The six statements on the left are facts (i.e. greg is mark's father). The three on the right are if-then rules. The last one can be interpreted as stating that X is a brother of Y provided that X is male and that both of them have the same parent. Through facts and rules like these, declarative knowledge about the world can be mapped directly into a programming language.
Given the above program, a very broad range of goals can be set and satisfied. Here are some examples of queries, with machine responses given on the right:

?- male(X).
   X = mark
   More (y/n)?
   y
   X = mike
   More (y/n)?
   y
   no

?- father(greg, X)
   X = mark
   More (y/n)?
   y
   X = mike
   More (y/n)?
   y
   no

?- parent(X, kathy).
   no

?- brother_of(mike, mark).
   yes

?- brother_of(mark, kathy).
   yes

Although it is the declarative nature of PROLOG that is important for us here, a corresponding procedural interpretation is always valid as well. For example, the "brother_of" rule above can be interpreted as stating: "To prove that X is the brother of Y, first verify that X is male, then find a parent of X who can be shown to be a parent of Y also."

A paramount feature of PROLOG is the clear distinction made between logic and control. Ideally, to write a program in PROLOG is to supply the body of logical knowledge and to map it into a set of facts and rules. The pattern-directed control mechanism in PROLOG is completely transparent to the programmer. (This is an idealized description; in reality PROLOG has some control predicates which disturb the purity of logic programming.)

Conventional procedural languages describe "how", in contrast to the "what" described by declarative languages such as PROLOG. Because every program must contain some knowledge, the PROLOG approach offers enormous advantages in the form of reduction of software development effort because of its modularity and embedded control mechanism, i.e. inference engine.
Let us briefly examine some PROLOG features:

- A nonprocedural approach, with a highly modular and modifiable structure;
- Uniform representation of program and data;
- A control mechanism based on unification/resolution and depth-first search with backtracking;
- A built-in DBMS.

A detailed description of PROLOG can be found in [1].

The remainder of this article is divided into three sections. Section 1 discusses the role of logic programming in mapping a system specification directly into code. Section 2 suggests a possible approach for building a "generic" software cost estimation (SCE) model, and Section 3 reviews the role that logic programming can play in introducing intelligence to the data input portion of the SCE model.
1. PROLOG and the System Specification

The software life cycle can be represented as a set of major phases which follow each other in time. Ignoring (for simplicity only) the first phase, feasibility analysis, we can start with the plans and requirements phase, also known as the system specification. This phase is a prerequisite to the design phase, which in turn is prerequisite to the programming phase.

To produce a design from a system specification is generally a lengthy and complex task, closely related to the practice of black magic. The effort devoted to producing the system specification is just the beginning: at least equal effort, and often much more, must be expended to produce a viable design. The design, moreover, is only a precursor to the actual programming.

The concept of "automated programming" was developed to address this situation. In a sense, it represents an effort to produce a black box which would convert a system specification into an implemented program. The results of such efforts have so far been rather discouraging.

Why is it that we face such a severe problem? The primary reason is that after providing a body of logical knowledge in the form of the system specification, we attempt to convert this knowledge into a procedural form through the design and programming processes. And why a procedural form? The only reasonable explanation of which we are aware is that the tools we have at our disposal, conventional programming languages, are oriented in a procedural rather than a declarative fashion.

Kowalski was the first to suggest the direct transformation of a system specification into a logic program [2]. To accomplish this, the system specification should be represented in some form of logic specification language. DeMarco Structured Analysis is an appropriate example of this [3].

Let's look at a subset from a case study in structured analysis, and indicate how it can be transformed into a corresponding PROLOG program. For those who are interested and have some familiarity with Data Flow Diagrams, the Appendix provides an example with some of the details of this transformation. The example is the "Astro-Pony Tootshops" operation, taken from Chapter 8 of DeMarco [3]. It has been simplified by omitting diagrams relating to financial processes.

The primary tools of Structured Analysis are:

Data Flow Diagrams

The Data Dictionary

Structured English
Data Flow Diagrams may be viewed as a network of interactive processes, each of which represents a distinct logical function, and associated data inputs and outputs, or data flows. The Data Dictionary describes the composition of the data flows. The full set of diagrams is produced by a partitioning procedure, which starts from a single top level diagram, the Context Diagram. At the lowest level, the unpartitioned processes, or functional primitives, are described in Structured English.

The transformation of a Structured Analysis into a PROLOG program is fairly straightforward. Each Data Flow Diagram and each descriptive statement in Structured English is represented as a PROLOG predicate. Elements of the Data Dictionary become PROLOG variables. In addition, the PROLOG code contains some other constructs related to file manipulation, selection processes, specific control structures, etc.

The most important feature of this mapping is its one-to-one correspondence between the system specification and the resulting PROLOG program. Compare this correspondence with an attempt to transform functional requirements into a conventional programming language! All key features of data flow diagrams, such as top-down design with the capability for further refinement, are reflected in PROLOG code. Modularity is provided by the fact that functions can be easily isolated and modified. The data flow diagrams on all levels but the lowest are represented by the declarative portion of the PROLOG code. Only on the lowest level, where the functional primitive is expressed through an appropriate Structured English description, is the procedural interpretation more appropriate.

Not only are the system specification and PROLOG code corresponding to it functionally the same, but they look almost identical! This similarity suggests that we may be able to automate the conversion procedure. We are thus back to "automated programming", but can now build on a more solid foundation. This is definitely the case for all levels of data flow diagrams above the level of structural primitives. Mapping the functional primitives themselves (i.e., a Structured English description) into PROLOG code will require much more effort.

The most important advantage of this approach is the replacement of the design and programming phases with a process which consumes much less time and effort. This is the basic reason why PROLOG is so well suited for fast prototyping. Beyond that, it is worth mentioning that we derive an additional benefit in the form of a convenient tool for dealing with the following difficult problems:

- completeness of mapping and the degree of completeness;
- traceability, or the capability of following requirements through the various levels of the specification;
- consistency, both static and dynamic.
2. Building a Generic SCE Model

With software development assuming an increasingly significant role, effective and reliable software cost estimation becomes critically important. The output of the SCE process is usually presented in the form of numerical estimates of required effort, expected costs, and efficient schedules. That is, the SCE answers questions like: "How much will it cost?", "How long will it take?", "How many people will we need at each stage of the project?", "What are the trade-offs?", etc. This output is critically dependent on the nature of the SCE model used, and also on the values assumed for input parameters.1

Progress in this area, however, is still very limited. There are two reasons for this:

- there are only a limited number of models available.
- there are only a limited number of commercial packages available which support these models.

The most troublesome issue is that of identifying an efficient set of critical input parameters which will be effective as a basis for estimation. Barry Boehm has categorized those discussed in the literature (about 35 of them) as follows:

- size attributes
- program attributes
- computer attributes
- personnel attributes
- project attributes

The best known SCE models, such as Boehm's COCOMO and Putnam's SLIM, use a large number of critical parameters. For these models, an accurate estimate of the eventual number of source instructions is very important. Because of this, they provide good estimates for many phases of the software development process, but not for the initial conceptual phase, where an accurate estimate of source instructions is difficult, if not impossible.

Recent research has proceeded along two parallel but independent lines:

Further refinement and incorporation of additional detail into existing models oriented around the number of source instructions. There are more than 150 different parameters which have been used to date, and the issue of which of them are the most significant has not yet been fully resolved.

---------------------------------------------------------------------
1 Henceforth, the terms "input parameters", "critical parameters" and "project attributes" will be used interchangeably.
Construction of alternative models whose objective is to achieve some measure of accuracy of estimation during the very early, i.e. conceptual, stages of the life cycle (see for example [6]). This is done by using as input parameters the values of the more tangible "business variables", such as input/output classes, number and types of files needed, functions to be performed, types and categories of systems, development and operational environment, etc., rather than parameters like number of source instructions which are completely unknown at this point. Despite the attractiveness of this approach, its validity as well as the identification and calibration of the most important critical parameters remain to be demonstrated.

We can see, then, that the primary problems involved in trying to formulate a generic SCE model are:

- unification of existing SCE models best adapted for use at different stages of the life cycle. This entails incorporation of enough intelligence to make choices among alternative estimating algorithms conditional upon the state of the estimating environment; and

- establishing an objective and effective mechanism for ascertaining the potential value of an attribute as a critical parameter, and for calibrating the parameter accurately.

The field of software cost estimation is still definitely in an exploratory stage -- a lot more experimentation is still needed. In this kind of development environment, the conventional methodology with its rigid skeleton consisting of functional specification followed by design followed by programming simply does not fit. Artificial Intelligence methods in general [7], and logic programming in particular seem to offer promising tools to fill the vacuum.

Building a generic SCE model using logic programming (PROLOG) as the primary tool will entail two main tasks:

Specify a set of distinct project attributes, with estimated degrees of uncertainty for each;

Formulate a set of rules by means of which the SCE outputs will be derived from the values of these project attributes or input parameters.
In other words, we need to identify a set of relevant facts and prescribe a set of rules for using those facts which reflect the appropriate body of logical knowledge, and which explicitly takes into account variable levels of combined rule/fact uncertainty. The set of candidate attributes for evaluation should be very broad ranging. It should include any characteristic which might convey significant information value under any likely circumstances. This would provide maximum leeway for experimentation and calibration by the developer, and later by the user.

The specification of uncertainty will provide a basis for discrimination. Conditional processing mechanisms would select among alternative algorithms using different subsets of the facts, thus ignoring information whose degree of uncertainty exceeds some threshold value.

Identifying a set of potentially valuable project attributes will be a lengthy, but relatively straightforward task. Formulating the set of rules will be more difficult, but as mentioned in Section 1, it should be done anyway, even for a conventional environment. We may as well conduct the formulation in an explicit, modular fashion, leaving ourselves almost unlimited leeway for experimentation and tinkering.

What do we have now? Believe it or not, we already have our generic SCE model! Because of PROLOG's modularity, once we have introduced the uncertainty factors we will be able to:

- establish a bridge in our model among different phases of the life cycle, because our rules will be context-sensitive, and the phase we are in will in part determine the context;
- provide very broad capabilities to determine the relevance of individual attributes, and to permit their calibration by the user through the adjustment of appropriate weight factors;
- provide a feedback mechanism enabling self-calibration on the basis of an evaluation of past performance of the model. This is a much more difficult task, entailing the general concept of "learning by experience".
- provide an explanation facility which would disclose the logic behind the results produced by the model.
In Section 1, we suggested that the final program can be implemented in the form of a logic program rather than a conventional program, and that this can be done directly from the system specification, with less effort and in a much more natural fashion. If this is indeed true, then it opens up entirely new horizons for the SCE. The role of "business" project attributes for the SCE at the conceptual stage might be assumed by parameters associated with logic programming, such as the numbers of rules and of facts. This could probably be done on a more unified and logical basis, hopefully with more reliable results. It is very important to ascertain the exact nature of these parameters if we want to have even a rudimentary SCE in an AI software development environment [8]. Moreover, these new parameters may even fully replace the ones currently used.

In summary, we believe that logic programming has the potential to make a significant contribution to the troublesome but important function of software cost estimation.
3. Intelligent Data Input Assistance

The issue of quality and integrity of data input is one which is assuming increasing importance. In situations where data input may be erroneous, inconsistent or incomplete, the function of input validation is no less important than the quality of the algorithms which actually process whatever input is provided.

There is ample opportunity for fruitful application of AI tools to assist in the function of obtaining valid input. The approach that we are pursuing is based upon the notion that an accumulation of experience on the part of the system receiving the input can be utilized by that system to review what it is receiving currently. The experience base may be either passive or active: in the former case it is restricted to simply retaining a memory of what it has seen before; in the latter, adjustments are made to that memory to reflect relative degrees of satisfaction with the results of input processed in previous situations. Thus far our work has been limited to the passive case, but we hope to move on soon to the active one.

This approach provides a good example of "learning by experience", albeit on the input side only, to which we alluded earlier. Among the capabilities it offers are:

- Validation of new user-provided information in much the same manner as is currently done by conventional methods;
- Identification and flagging of instances of user-provided input which are inconsistent with a standard that is dynamically determined by past experience. The nature of the inconsistencies flagged, as well as the threshold deviation for flagging, can be easily changed without reprogramming;
- Suggestion of input for consideration by the user, based on what has been used frequently in similar prior situations.

Exactly what constitutes a "similar situation" is difficult to define in a completely satisfactory fashion, and the value of the facility is effectively determined by the degree of success achieved in producing that definition. Given a satisfactory context-specific definition of similarity, however, this framework is quite general and can be utilized in many different kinds of applications. We have used it in a package called the GHL Strategic Planner (9). That system relies strongly upon the quality and completeness of user-provided input, and the input forthcoming is likely to be flawed because the user is dealing with projected or even hypothetical events. In such situations, imagination necessarily plays a dominant role. While often powerful, however, imagination is not very reliable.
Similar characteristics are found in the SCE application, and therefore the quality of SCE input parameter values might also be improved by the consideration of previous experience. Clearly, the road is open for a variety of fruitful development efforts.
References


APPENDIX

Figure 1
Figure II

race_sheet

birth_report

1. RESEARCH

RACE FILE

HORSE FILE

2. SALES

3. ASTROLOGY

order invoice profile
Figure III

1.1 ENTER NEW RACE

1.2 RECORD BIRTH

race_sheet

birth_report

HORSE FILE

RACE FILE
Figure IV

RACE FILE

2.2 MATCH RACE

race_specific order

reject

2.1 CHECK VALIDITY

order

reject

general order

matched_race_specific_order

2.3 PRODUCE INVOICE

invoice
Figure V

3.1 DETERMINE CLASS OF SERVICE

3.2 SELECT BEST RACE

3.3 FILL ORDER

HORSE FILE

RACE FILE

Profile

invoice

general invoice

specific invoice

partially filled order
Functional Primitive Example

FILL ORDER (3.3):

For each arriving order (partially filled order or specific invoice), do all of the following:

- Pull Race File folder for the race.
- Look up Julian day number of Customer_Birth_Date.
- For each horse in the Race File folder:
  - Calculate the difference in days between Horse_Birth_Date and Customer_Birth_Date.
  - Express the difference as a positive number.
- Select horse(s) for which the difference is closest to 23.
- If more than one horse is selected:
  - Order the selected horses alphabetically by Horse_Name.
  - Select the last one.
- Make up a prediction card with Horse_Name and Horse_Astrol_Sign.

Selected Data Dictionary Elements

order = Cust_Name.Addr + Customer_Birth_Date + (Race_Specifier).

invoice = Invoice# + order + Salesman_ID

profile = invoice + Custom_Prediction

Race_Specifier = Race_Date + Track_Name + Race_Number

Custom_Prediction = (Race_Specifier) + Horse_Name + Horse_Astrol_Sign + Cust_Astrol_Sign

Race.File = ( Race_Specifier + Horse_Name )

Horse.File = ( Astrological Data about each Animal )
**PROLOG Code for Upper Levels**

\[\text{apt(} \text{Order, } \text{Profile):=} \]
\[\text{ sales(Order, Invoice),} \]
\[\text{ astrolology(Invoice, Profile).} \]

\[\text{ sales(Order, Invoice):=} \]
\[\text{ check_validity(Order, X),} \]
\[\text{ belongs_to_order(X, race_specific_order),} \]
\[\text{ match_race(X, Matched_race_specific_order),} \]
\[\text{ produce_invoice(Matched_race_specific_order, Invoice).} \]

\[\text{ sales(Order, Invoice):=} \]
\[\text{ check_validity(Order, X),} \]
\[\text{ belongs_to_order(X, general_order),} \]
\[\text{ produce_invoice(X, Invoice).} \]

\[\text{ astrolology(Invoice, Profile):=} \]
\[\text{ determine_class_of_service(Invoice, X),} \]
\[\text{ belongs_to_invoice(X, general_invoice),} \]
\[\text{ select_best_race(X, Partially_filled_order),} \]
\[\text{ fill_order(Partially_filled_order, Profile).} \]

\[\text{ add_to_horse_file(Birth_report):=} \]
\[\text{ record_birth(Birth_report, X),} \]
\[\text{ assert(X).} \]

\[\text{ add_to_race_file(Race_sheet):=} \]
\[\text{ enter_new_race(Race_sheet, X),} \]
\[\text{ assert(X).} \]

**PROLOG Code for Some Functional Primitives**

\[\text{ enter_new_race(Race_sheet, Out):=} \]
\[\text{ retrieve_horse_file_rec(Race_sheet, X),} \]
\[\text{ build_new_record(Race_sheet, X, Out).} \]

\[\text{ fill_order(X, Profile):=} \]
\[\text{ convert_cust_birth_date(X, C_Jul_day),} \]
\[\text{ build_horse_list(H_list),} \]
\[\text{ build_horse_differ_list(C_Jul_day, H_list, HD_list),} \]
\[\text{ sort_above(HD_list, SHD_list),} \]
\[\text{ select_horses_closest_to23(SHD_list, 23_list),} \]
\[\text{ select_one_horse(23_list, Horse_name, Horse_Astrol_Sign),} \]
\[\text{ finalize(X, Horse_name, Horse_Astrol_Sign, Profile).} \]