Prolog at IBM: An advanced and evolving application development technology

by M. Bénichou
H. Beringer
J.-M. Gauthier
C. Beierle

Prolog is a powerful programming language, based on logic, that originated and matured in Europe. This paper aims to show that Prolog is becoming one of the key tools for the entire application development community. First explained is how the unique properties of Prolog give it many advantages over classical languages. Then we show that the language is sufficiently mature technically so that numerous industrial Prolog products are now available. In particular, IBM offers the Systems Application Architecture® (SAA™) AD/Cycle™ Prolog product family, which provides a combination of logic programming and object programming facilities. Many industrial applications are written in Prolog. Examples of 15 outstanding operational applications, developed by IBM or major IBM customers, are presented. There is a potential for future growth in the types of applications enabled by improving the language. The simplicity and elegance of the theoretical basis of Prolog allow a number of extensions to be defined. Here, three European projects are briefly presented. In conclusion it is shown that Prolog, possibly extended in many directions, is one of the tools that could help solve the long-standing quality and cost problems in application development.

In the early 1970s, using the work done by J. A. Robinson on problem-solving in logic,[1] Alain Colmerauer and Philippe Roussel (professors at the University of Marseilles) along with Robert Kowalski (at that time professor at Edinburgh University) first defined Prolog.[2,3] Its very name is a reminder that it falls within the theoretical framework of logic programming (*PROgrammation en LOGique* in French). This programming language was originally intended as a basic tool for the difficult tasks of analysis and interpretation of natural languages. Research continued, and in the late 1970s David Warren implemented the first efficient compiled version at Edinburgh University.[4]

It was in 1981, however, that Prolog took the computing world by storm when it was chosen by the Japanese as the basic language for their Fifth Generation Project. Even though this project produced disappointing results regarding the implementation of Prolog, perhaps because it was too heavily concentrated on the development of dedicated Prolog machines, the number of people knowing, appreciating, and using Prolog has been constantly increasing. Prolog is now recognized as a major programming language. In the United States Prolog is becoming a standard alongside LISP. Prolog is taught in most universities (par-

ticularly in nearly all European ones) and has become a major vehicle for doing research in computer science.

Very early on in Europe, several IBM Scientific Centers became interested in Prolog. Existing IBM Prolog products originated at the Paris Scientific Center where, under the aegis of Marc Gillet, various prototypes were developed that have since been transformed by IBM laboratories into official products, development now being the responsibility of the Paris Scientific Center.[5] This work has also inspired many theoretical and applied research projects at the Paris Center.[6–9] Logic programming has also been the basis of significant projects[10,11] at the German Scientific Center (the Heidelberg Scientific Center and the Institute for Knowledge-Based Systems). But the research on Prolog in IBM is not limited to the European Scientific Centers; very important work has been carried out worldwide, especially in various Scientific Centers and the IBM Research Division.

In industry, Prolog has been used for many years, first as the language of artificial intelligence, especially in natural language processing, and now in more general applications. This paper shows that, thanks to major advances in the implementation of Prolog compilers, even very large size applications can be developed entirely in Prolog, from the initial prototype up to the final operational product.

Today, Prolog is moving from the world of academia and research into industry and the service sector: There are dozens of commercial Prolog implementations on all types of platforms, and Prolog has been chosen for well over a thousand operational applications, in particular in the manufacturing sector. It is already beginning to be used in pure business applications, and this use will no doubt increase further. The Prolog Vendors Group was recently formed by the leading Prolog suppliers. Its goal is "to widen the market for Prolog technology."

An objective of this paper is to demonstrate that Prolog is becoming one of the key tools for the entire application programming community.

For the content of the application and research parts we appealed to a number of people both in IBM and outside of IBM who gave us valuable, first-hand information. Their names are ascribed to the parts to which they contributed and are listed in the notes at the end of the paper.

Unique features of Prolog

Prolog differs from other programming languages in that it offers a completely new approach with many advantages. In particular, it has the following properties:

- Prolog is *declarative*; programming in Prolog means simply writing *logic* statements, which can be done without having to worry about any problem-solving algorithms.
- Prolog is a *relational* programming language that handles queries having multiple solutions without asking the programmer to manage them. This aspect is called *nondeterministic programming.*
- Finally, Prolog is a *symbolic* programming language. Symbols and complex symbolic expressions (i.e., any tree) are easily handled in Prolog, thanks to a simple but powerful mechanism: *unification.*

In order to illustrate these statements, a quick description of the basic concepts of Prolog and a short example are given below. The interested reader will find an advanced introduction to Prolog in Wilson[12] and a complete description in Walker et al.;[13] programming with IBM Prolog is explained in detail in Reference 14; examples of the Prolog use, classified by problem types, can be found in Yoder.[15]

Basic concepts of Prolog. Prolog handles relations among objects. These objects are represented by symbolic expressions, called "terms," which may be:

- Constants such as numbers and names like "`paris`" or "`france`". (They begin with a lowercase letter.)
- Functional terms written "`function_name (arg1, . . . ,argN)`" where the arguments may be any term. For example, the term "`time(8,30)`" may be used to represent the time 8:30.
- Lists written "`[elt1,elt2, . . . ,eltn]`" or "`[elt1,. . . ,eltn|List_end]`", their elements being any term. The empty list is the constant "`[ ]`". (As a matter of fact, lists are just a special case of functional terms with some syntactic facilities.)

A variable (beginning with an uppercase letter) may be used in place of an unknown term in exactly the same way as mathematical variables. For example, `time(8,Min)` could represent a time between 8 and 9 a.m. depending upon the value of `Min`. It should be stressed that a Prolog variable is a logic variable representing a single unknown term. When the corresponding term is known, it definitively replaces the variable.

Unification is an operation that tries to make two terms equal by substituting their variables. If this is not possible, the operation fails. Surprisingly enough, this operation is the only one available in Prolog to test terms as well as build or transform them. As an example, by trying to unify the term `time(H,M)` with a term `T`, one first tests whether `T` may be made equal to a functional term `time` with two arguments, otherwise unification fails. Now, if, for example, `T` is `time(8,30)`, unification succeeds and `8` and `30` are substituted for the variables `H` and `M` respectively. These two variables may then be used to build other terms such as `appointment(monday,H,M)`.

Relations among objects (or "predicates") are identified by a name that is a character string beginning with a lowercase letter and by a fixed number of arguments written between parentheses. A predicate can be defined by a list of facts exactly as a table defines a relation in a relational database. The predicate `in_country (Town,Country)`, for example, meaning that `Town` is in `Country` may be defined by the facts:

```
/*          Town     Country  */
in_country(paris,   france).
in_country(nice,    france).
in_country(rome,    italy).
```

In the same way, a database about flights in Europe can be entered as facts defining the relation `flight(FlNum,Origin,Dest)`, where `FlNum` is the number of the flight and `Origin` and `Dest` are the towns it connects:

```
/*     FlNum Origin  Dest  */
flight(531,  paris,  nice).
flight(220,  paris,  nice).
flight(121,  paris,  rome).
```

A Prolog program is a sequence of first-order logic formulae of a specific kind called (Horn) clauses. A clause is either a simple fact as above or a rule made up of two parts, a head and a body. The head, which is the conclusion of the clause, is a predicate with its arguments. The body, which forms the premise, is a conjunction of predicates with their arguments. A clause is written in the following way:

```
⟨head⟩ :- ⟨body⟩.
```

and reads, if ⟨body⟩ then ⟨head⟩.

A clause (rule or fact) may contain variables that are implicitly universally quantified.

As an example, if we want a program able to find domestic flights in a given country, it suffices to state the following logic rule:

```
FlNum refers to a domestic flight in Country if:
      FlNum refers to a flight from Origin to Dest, and
      Origin is located in Country, and
      Dest is located in the same Country.
```

This rule is naturally translated into a Prolog clause as follows:

```
domestic_flight(FlNum, Country) :-  /* if: */
      flight(FlNum, Origin, Dest),  /* and */
      in_country(Origin, Country),  /* and */
      in_country(Dest,   Country).
```

Together with the above facts, *this rule is a complete Prolog program* able to answer many different queries such as:

1. Is flight 531 a French domestic flight?
2. Is flight 531 a domestic flight and if so in which country?
3. Which flights are French domestic flights?
4. Give me all of the domestic flights together with their countries.

Moreover, these four queries are asked simply by entering the appropriate relation together with its arguments that may be either known or unknown. For example, the third query above is entered as:

```
?- domestic_flight(FN, france).
```

where the symbol `?-` indicates that it is a query, including here an unknown, `FN`. Prolog gives one answer (`FN=531`) and then backtracks, i.e., searches for an alternative solution, as a result of which, it gives a second answer (`FN=220`). To find

this second solution, Prolog tried to reconsider one of its preceding choices, in this case the fact used to solve the subgoal `flight`.

The core of Prolog is as simple as that. However, in spite of (or because of) this simplicity, Prolog has proved to be a very powerful programming language. To make Prolog a "real" programming language, various practical extensions, called "built-in" predicates, must be added. These built-in predicates allow such basic operations as input-output, arithmetic, or string handling, offer various event or error controls, provide multiple interfaces (to classical languages, to databases, to graphics), etc. For instance, AD/Cycle Prolog has more than 300 such built-in predicates that have been incorporated, preserving the fundamental properties and the simplicity of the language.

The Prolog proof mechanism is powerful enough to accept relations being defined recursively.

In the very short example given above, the three basic properties of Prolog could be observed. We now take a closer look at them.

Prolog is declarative and modular. Prolog programs can be looked at from two different points of view, declarative (as in our example) and procedural.

From a declarative point of view, the meaning of a program is simply the set of all the facts that logically follow from the rules and facts that make up the program. Thanks to these very straightforward declarative semantics, writing and checking Prolog programs is easy because it comes down to writing logic assertions and checking their veracity.

From a procedural point of view, the meaning of a program is the succession of steps the Prolog inference engine will follow to try to prove queries, using the rules and facts of the program. To do so, Prolog uses a simple strategy: depth first and left to right, i.e., facts and rules are used in the order in which they have been written. It follows that a programmer can easily predict the successive steps of a proof and may control them if so desired.

It can be shown that the two views coincide. As a result, the Prolog programmer can easily switch to the point of view most suited to the current concern.

Since a Prolog program is structured in rules, it follows that a program is in fact made up of small declarative modules, i.e., sets of rules that define some relations, which can be tested separately and reused.

The Prolog proof mechanism is powerful enough to accept relations being defined recursively, which is often the shortest and most natural way of expressing relations concerning structures that are themselves recursive, such as lists, graphs, etc. Moreover, Prolog compilers are able to drastically optimize recursive programs.

Prolog allows nondeterministic programming. It is well-known that many algorithms are better expressed nondeterministically, in other words through the systematic examination of a set of possibilities, or through trial and error (trying to find one's way out of a maze, for example). Prolog contains a built-in mechanism to handle nondeterminism through a depth-first search and backtracking. This mechanism makes the implementation of such algorithms much easier, since Prolog takes over the entire management of the search tree. The mechanism can be controlled by the user, who can thus ask Prolog for the first, then the next or all solutions, tell Prolog in what order the search should be carried out, or even cancel the backtracking mechanism completely.

Symbolic processing. Prolog is particularly well-suited to the processing of complex symbolic structures. Since the basic objects it handles are trees, it is very easy to represent attribute-value sets, variable size lists, graphs, etc. Through simple unification these trees can be built, read, and transformed without the user having to worry about how they are actually represented in memory or having to consider problems of allocation, pointer handling, etc. Prolog takes over the dy-

namic management of the memory (up to the recovery of unused memory).

Prolog is therefore particularly suitable for the analysis, translation, and generation of formal and natural language (which is hardly surprising since this is precisely what it was developed for). Moreover, it is also very easy to write Prolog programs that analyze, generate, and execute Prolog. (This is called meta-programming.)

Prolog: A mature technology

Prolog is attractive not only because of its nice theoretical foundations and properties, but also because of the high quality of the Prolog compilers now on the market. In particular, IBM offers a Systems Application Architecture* (SAA*) AD/Cycle* Prolog product family that is a complete implementation of the Prolog language plus several powerful extensions. The two existing products are AD/Cycle Prolog/MVS & VM Version 1 (5696-308) and AD/Cycle Prolog/2 Version 1 (5696-309).[16-18] These are new implementations of earlier products: IBM Prolog for 370 (5706-236) and IBM Prolog for Operating System/2* (OS/2*) (5621-065).

The AD/Cycle Prolog development environment encourages productivity by providing a consistent means of expressing all of the phases of application development within the AD/Cycle framework, using the AD/Cycle workstation platform (WSP) and conforming to AD/Cycle Level 2 integration.

The AD/Cycle Prolog Extension Features[19,20] add a number of facilities to Prolog, including object-oriented ones. This combination of logic programming and object programming facilities provides a unique application development productivity tool. (This is a new implementation of earlier IBM EMEA program offerings, IBM Prolog language workbench [IPW] for 370 [5787-AAF] and for OS/2 [5776-ABH]; the extension feature is based on EMICAT, a registered trademark of Dassault Electronique.)

AD/Cycle Prolog. AD/Cycle Prolog contains an interpreter, an incremental compiler, and a full compiler that interact well together in order to provide high performance for large-size applications. In addition, programs are *fully portable* across the virtual machine (VM), Multiple Virtual Storage (MVS), and OS/2 operating system platforms. AD/Cycle Prolog also offers all of the features to be found scattered throughout the other classical industrial Prolog packages. AD/Cycle Prolog can *integrate applications* in a company's information system, thanks to powerful interfaces with the operating systems, editors, databases, dialog and graphic managers, and classical languages.

A source level interactive debugger allows a user to follow the execution of his or her program in the source code; under OS/2, it provides multiple windows and an interface to external source editors.

The Prolog language features include various data types (arrays, items, big rational numbers, extended skeletons, and user-defined types), delayed evaluation, global terms, error recovery, logical interrupts, external predicates (e.g., defined in C), computable expressions, automatic garbage collection, national language support, and double-byte character set.

Under VM and MVS, AD/Cycle Prolog provides code sharing and uses Extended Architecture (XA), allowing applications that reside in address spaces above the 16-megabyte line to be run. Under OS/2 2.0, it runs in 32-bit mode and provides a full Presentation Manager* (PM) interface, thus allowing the handling of PM user windows using Prolog predicates.

The CICS/IMS subsystem feature enables an MVS subsystem that controls Prolog servers running Prolog applications to be created. The servers can be called from Customer Information Control System (CICS*) or Information Management System (IMS) user transactions or can in turn call back CICS or IMS with requests for terminal access and data management.

The Extension Features. The AD/Cycle Prolog Extension Features (EFs) also run under MVS, VM, and OS/2. They offer:

- *Object-oriented facilities*, supporting objects, attributes, inheritance, demons, methods, and message-passing
- *Knowledge base management*, providing the possibility of carving up the Prolog workspace into pieces, called knowledge subsets (KSSs), and saving only those pieces. KSSs are portable between various platforms (VM, MVS, and OS/2)

and help to organize the developer's knowledge base, thus allowing multiple programmers to work on it.

- *Presentation Manager window support* in which object-oriented programming is especially useful for graphical user interfaces, as they tend to be based on specific objects (windows, buttons, etc.) that respond to specific "messages" directed to them (e.g., a mouse click). EF provides a set of predefined visual objects that can be used to quickly create and refine user graphic interfaces.
- *Rule-based systems creation*, providing facilities to easily develop rule-based applications and an environment for the easy handling of rules expressed in an external user-defined syntax and formalism
- *Utilities and productivity tools*, a library of commonly used Prolog predicates

Being an *extension* of AD/Cycle Prolog, EF does not in any way mask or hinder any AD/Cycle Prolog facility or interface.

Deploying applications using AD/Cycle Prolog. There are several ways of deploying applications that were developed using AD/Cycle Prolog.

AD/Cycle Prolog/2 is the most convenient environment for developing applications, although AD/Cycle Prolog/MVS & VM also offers VM and MVS development facilities. An application developed in the OS/2 environment may be ported and executed on different hardware and software configurations:

- On a Personal System/2* (PS/2*) stand-alone system
- On a PS/2 local area network (LAN) system with or without a client-server facility
- On a host with nonprogrammable workstations supported by the VM/CMS (Virtual Machine/Conversational Monitor System), MVS/TSO (Multiple Virtual Storage/Time Sharing Option), MVS/CICS, or MVS/IMS. This configuration makes Prolog applications suitable for customers with large networks supported by CICS or IMS.
- On a host with PS workstations with a Prolog distributed application and with a host DATABASE 2* (DB/2*) database or with a distributed relational database. Such an application may be combined with host MVS/TSO batch applications developed using AD/Cycle Prolog as well.

Prolog: A valuable tool for application development

In many application fields, logic programming has proved to be a useful development technology. Some of these fields are listed below, and for each of them, some outstanding projects developed using IBM Prolog are mentioned. Most of these projects would have been uneconomical using classical languages and were viable thanks to Prolog.

1. *Software engineering*—Following are several outstanding examples of Prolog being used for the development of large-size workbenches (more than 50 000 lines of Prolog):

   - ADW**,[21] an integrated Computer-Aided Software Engineering (CASE) tool that supports the development life cycle of business applications, developed by KnowledgeWare, a U.S. software house. The Knowledge Coordinator, a key component of ADW, is entirely written in Prolog.
   - OCEANIC,[22] an insurance-oriented package developed by SOCS, a French software house
   - CASE/390,[23] an interactive environment for the development of MVS components, developed by the IBM MVS laboratory (Mid-Hudson Valley Programming Laboratory)

   However, Prolog has also been used for the development of more specialized but equally ambitious software engineering tools such as:

   - ALIEN, an integrated system for computer-aided writing of technical manuals (Dassault Electronique, France)
   - ADEPT, for the automatic generation (and verification) of test cases (IBM San Jose Laboratory)
   - AnDes, for the visual verification of external specifications of certain systems (IBM German Application Development Laboratory, Böblingen)

2. *Knowledge-based systems*—Prolog allows the development, at reasonable cost, of expert systems that cannot be reduced to the simple rule formalism available in commercial shells. Operational examples are:

- CATS/DIANA,[24–26] a model-based diagnosis system for analog electronic circuits. CATS/DIANA is the result of a joint study between Dassault Electronique and the IBM Paris Scientific Center.
- GIBUS,[27] for the management of low earth orbit satellite batteries (Dassault Electronique, under contract to the European Space Agency)
- DR2,[28,29] a diagnostic system in the field of semiconductor engineering (IBM Sindelfingen plant)
- ESFA,[30,31] a computer-aided integrated circuit design tool (IBM La Gaude Laboratory)

3. *Natural and formal language processing*—Natural, natural-like, and programming language processing are ideal applications to use Prolog, which is a near-perfect tool for efficient and fast implementation and was, after all, initially invented to handle precisely such problems. For instance, KALIPSOS[7,32–34] is an outstanding example of automatic understanding of French texts (IBM Paris Scientific Center). This research is in the process of being transformed for use in industrial applications.

4. *Intelligent management of databases*—Listed here are but two of the tools that have been developed using Prolog to extend the usability or the functions of relational databases. Prolog allows easy handling of both knowledge about the database structure and the grammar of a powerful and user-friendly query language.

   - The first tool, IBM SAA Language Access,[35–37] is an IBM program product that provides a natural language interface to a relational database and could equally as well have been put in the section on natural language processing (IBM Nordik Laboratory).
   - The second tool, SYLLOG,[13,38–41] is an expert database system shell prototype that has been built over IBM Prolog and the Structured Query Language (SQL) Database Management System (IBM Thomas J. Watson Research Center).

5. *Intelligent control of algorithmic programs*—The basic aim here is to use Prolog to introduce expertise or intelligence in applications or programs that already exist in a company. Whereas, initially, a human expert was required to control the running of a set of programs, the idea is to replace this person by a sufficiently "intelligent" Prolog program that will automatically set the different parts of the programs in motion according to the results obtained in previous stages. An operational example of this kind is APACE, using Prolog, to design a gear box (Peugeot SA, France).

6. *Heuristic resolution of combinatorial problems*—The implementation of algorithms for solving combinatorial problems such as scheduling or distribution has been made much simpler using Prolog. The advantage of this language is that its "nondeterminism" allows the different parts of the solution space (search tree) to be explored successively, using backtracking. An operational example is JOSHUA,[42] a journey scheduler used at IBM France for organizing administrative or technical symposia.

Logic programming: An evolving technology

Logic programming naturally lends itself to many extensions that will enable more and more applications to be profitably tackled with this technology. Indeed, a large number of research projects are currently under way. In this section, three European research and development projects are described. These projects have the following important practical implications:

- Efficient use of new parallel hardware architectures
- Extended expressive power to cover new application areas
- Easier implementation of large-size applications (software engineering) using Prolog

Or-parallelism for ProLog by BIM

The content of this subsection was contributed by A. Marien, L. Maes, and J-L. Binot, BIM, Belgium. BIM, a privately held Belgian group specializing in information technology, is the maker of ProLog by BIM, which has recently been ported on IBM RISC System/6000 machines and is being marketed in Europe by IBM under the trademark Prolog for AIX/6000.

Parallel hardware architectures, including both shared memory machines and distributed architectures, are now proposed by many hardware manufacturers.

Logic programming languages like Prolog have a large potential for parallel execution. The two main kinds of parallelism in Prolog are or-parallelism and and-parallelism. An *or-parallel* Prolog system is one that explores in parallel the alternatives for solving a single goal. *And-parallelism* tries to solve several goals simultaneously. Several strategies are possible with regard to the interdependence of the solutions of the goals. They range from all-solution-and-parallelism, which solves all goals concurrently and computes the solution as the intersection of the independent solutions, through stream-and-parallelism and goal-suspension-and-parallelism executing the goals, when the input arguments become instantiated, to goal-independence-and-parallelism, where only independent goals are executed concurrently. Other strategies of parallelizing Prolog are lazy evaluation and parallel execution of unification. So far, however, this dynamic research and development activity[43,44] has not resulted in industrial implementation of parallel Prolog systems.

BIM, in cooperation with SICS (a leading Swedish research institute in the field of logic programming), is now completing a coarse-grained or-parallel implementation for shared memory machines, using the MUSE (MUlti-SEquential) scheduler developed by SICS to control parallelism. The project, with the internal code name BIMUSE,[45,46] is being developed as part of the ESPRIT project PEPMA.

In order to achieve the coarse-grained or-parallel behavior, several conventional Prolog engines (ProLog by BIM) are coupled to a MUSE scheduler that distributes work to the Prolog processes, called "workers" below. The central idea is the notion of several independent workers each having its own local and some global memory.

The two main functions of the scheduler are to maintain the sequential semantics of Prolog and match idle workers with available work, with minimal overhead. Communication between workers is kept to a strict minimum and is mainly done through a shared search tree. The tree has two kinds of nodes—shared and private—representing the choice points. The root of the tree, being shared by several workers, is managed by the scheduler. The leaves with the private nodes are only accessible to their creators. The shared part of the tree grows when a worker makes its private nodes shareable and shrinks when the last worker backtracks from a shared node.

When a worker runs out of work within its private subtree, it enters the shared search tree and calls the scheduler, looking for more work. The scheduler then, basically, looks for workers with excess work to share, and requests sharing of nodes or, if no work can be found, positions the workers in the tree so that they can be reactivated with very little overhead.

To extend ProLog by BIM for this or-parallel execution model, only a minimal part (less than 1 percent) of the code had to be rewritten. Needed adjustments included memory management, calling the scheduler at the proper times, maintaining extra engine-specific data, and delivering the data when the scheduler requests it.

Initial benchmarks on a first version of the BIMUSE system indicate that the overhead is less than 10 percent and that the speedup is close to the number of additional processors for problems with a high degree of nondeterminism.

Constraint logic programming at Dassault Electronique

The content of this subsection was contributed by B. Botella and P. Taillibert, Dassault Electronique, France. Dassault Electronique, which has already made a name for itself at the forefront of electronics, has also become a major information processing and software company where Prolog plays a key role (over 150 people have been trained to use Prolog; more than 50 of them actually use it every day).

Solving the physical problems found in the industry requires an ever-increasing use of artificial intelligence techniques such as model-based reasoning, for which constraint logic programming (CLP) is a useful implementation technology. This recent development of logic programming allows a programmer to declare constraints on the variables that are handled by Prolog. For example, it is possible to write, once and for all, that "X = Y + 2" and leave it up to the inference engine to do the rest and, as soon as a value is known for X or Y, to compute the other value and propagate the consequences.

The CLP package INTERLOG is the outcome of exploratory work carried out by Dassault Electro-

nique along the lines of the work of the Bell Northern Research laboratories. It extends IBM Prolog with the ability to handle constraints on real intervals. INTERLOG is especially appropriate to the modeling of physical phenomena in mechanics or electronics. As a matter of fact, these physical phenomena can only be described using imprecise or nonlinear models, and interval arithmetic is an adequate tool to handle these models. As such, INTERLOG can benefit from the powerful logic programming facilities offered by IBM Prolog and extend them to handle numeric problems. The first version of INTERLOG under VM and OS/2 is operational at Dassault Electronique and has been used successfully both in computer-aided design to solve a hitherto unsolved problem and in an expert system for computer-aided design.

Research and applications in PROTOS, a EUREKA project

The content of this subsection was contributed by C. Beierle, IKBS, Stuttgart, IBM Germany, H. Beringer, Paris Scientific Center, IBM France, and J. Jachemich, Hoechst, Germany.

The objective of the EUREKA project PROTOS ("Logic Programming Tools for Scheduling Applications," EU56) is twofold: the development of various extensions of logic programming for building knowledge-based systems, and the use of these extensions in production planning and scheduling applications. The Institute for Knowledge-Based Systems (IKBS) of the IBM Germany Scientific Center is taking part in this project in close collaboration with the IBM Paris Scientific Center, together with the project partners Hoechst AG (Germany), Sandoz (Switzerland), BIM (Belgium), and the German universities of Bonn and Oldenburg.[47]

In the following subsections, we begin by describing how Prolog can be given a useful and powerful constraint-solving ability and what work is being done on the constraint extension of IBM Prolog. We go on to consider various original features such as types, modules, and deductive database access of the logic programming language PROTOS-L,[10] which is one of the concrete results of the research effort within PROTOS. Finally, we describe one of the major applications developed in PROTOS, a knowledge-based production planning support system that was developed in close collaboration with a project partner, Hoechst.

Constraint logic programming. Although Prolog is very powerful for handling symbolic expressions, it is limited when dealing with objects of a structured domain such as real arithmetic. For instance, suppose a database about cars is described by the relation `car(Model,NbPlace,Price)` in the following way:

```
/*  Model    NbPlace   Price */
car(205,       4,      50000).
car(bmw,       5,     100000).
car(ferrari,   2,    1000000).
```

Prolog is able to answer queries such as:

- How many seats does the 205 have and how much does it cost? `?-car(205,NP,P).`
- Which cars are four-seaters? `?-car(Model,4,P).`
- Which cars cost less than 60 000 Francs (F)? `?-car(M,NP,P), P<60000.`

However, the price of a car is not fixed and may vary according to options, special offers, etc. For example, a 205 may cost between 50 000 F and 80 000 F. Dealing with those kinds of imprecise data is no simple matter. In order to describe the price range of a 205, it would be natural to replace the first of the above facts by:

```
car(205, 4, P) :- 50000≤P, P≤80000.
```

However, in standard Prolog, this does not work; the three queries above would all result in an error message. As far as numerical computations are concerned, Prolog offers nothing more than what a traditional programming language offers: evaluation of expressions without unknowns.

In the CLP($\mathbb{R}$) language,[48] the three queries work normally with the clause modified as above. CLP($\mathbb{R}$) is one example of the languages entering the general CLP scheme, the purpose of which is to add arithmetic relations and functions to Prolog without losing any of the qualities of Prolog.

Genesis of constraint logic programming. In 1983, the Prolog II language developed by Colmerauer and his team[49] already offered some extended facilities (in particular, a primitive to force two variables to have different values). In 1987, Jaffar and Lassez of the IBM Thomas J. Watson Research Center[50] showed that this extension, like many others, could be described as a special

case of a unique framework: constraint logic programming (CLP).

The purpose of this framework is to give Prolog the ability to handle objects, functions, and relations of a specific domain (e.g., real arithmetic) according to their algebraic properties while preserving the fundamental properties of logic programming, especially its declarativity. Having observed that Prolog unification is nothing but an algorithm to solve equalities between symbolic terms, Jaffar and Lassez proved that it was possible to replace this unification by a more powerful algorithm (called a constraint solver), solving not only equalities but also any kind of constraint over the chosen domain. The resulting framework is especially simple and may be used for many domains providing they have two simple properties, as pointed out in Jaffar and Lassez.[50]

Nowadays, several CLP languages exist, including:

- The Prolog III language of PrologIA,[51] which deals with constraints on Boolean formulae, rational arithmetic, and lists
- The CLP(ℝ) language from the IBM Thomas J. Watson Research Center,[52] which handles constraints in real arithmetic
- The CHIP language of the ECRC,[53] which solves constraints on Boolean formulae, rational arithmetic, and integer arithmetic

Many other domains have also been imbedded into CLP prototypes, from polynomial constraints[54] to complex symbolic constraints.

Applications of CLP. Whether using a traditional programming language or Prolog, when some arithmetic is needed, the programmer has to explicitly describe computations while mastering the information flow (the meaning and the status of a variable at a specific point in the algorithm). When the system acquires information in an unspecified order, the same arithmetic relation (or constraint) must be translated into several different computations throughout the algorithm, depending on which variables are known. Moreover, if partial information (as for the price above) must be used, data structures and computations become highly complex.

Therefore, as soon as either incomplete information or an unordered information flow has to be dealt with, CLP is a valuable tool. The following are some examples of such problems.

Configuration systems with CLP. Configuration systems are tools helping an end user choose a specific configuration of devices and features while following some compatibility rules. These rules are easily expressed as constraints.

CLP is a valuable tool for incomplete information or for an unordered information flow.

The basic step in the configuration process is as follows: Given the partial configuration selected so far, the existence of a corresponding valid configuration must be verified, and all of the consequences of current choices have to be inferred.

With a CLP language, it is the constraint solver that performs this basic step. It is therefore easy to develop very flexible configuration systems with which the user may enter choices in any order. Moreover, choices may be expressed very freely as constraints. For example, in a microcomputer configuration system, the user could request a machine with more than 16 Megabytes of internal memory, the set of possible machines being automatically reduced.

Such a configuration system has been developed by Bang & Olufsen, a Danish hi-fi company, with the CLP(B) prototype (see below).

Device simulation, verification, and diagnosis with CLP. Many physical devices such as electronic circuits or mechanical structures may be modeled by constraints between their parameters. With a constraint-solving capability, the *same* model can then be used for the following:

1. Simulation: Given inputs, outputs are automatically deduced.
2. Verification: The constraint solver can check

whether the device model implies certain desired properties (expressed as constraints).
3. Diagnosis: Given some observed discrepant behavior, a search for the misbehaving component(s) can easily be implemented in CLP, taking advantage of the constraint solver to control this search.

Solving combinatorial problems with CLP. A combinatorial problem is one of finding an (optimal) combination of features that is consistent with some constraints, the number of possible combinations being enormous. Examples of such problems are: complex allocation problems, scheduling (with resource constraints) in manufacturing, distribution, transportation, etc., timetable generation, and many decision problems.

These problems are easily expressed in a CLP language offering constraints on variables belonging to a discrete domain (e.g., Boolean or integer). However, they are difficult to solve (i.e., NP-complete), and, in general, the constraint solver cannot perform more than a partial consistency test (without being exponentially slow). So, the overall problem is solved using an implicit enumeration strategy specified by the user. Implicit enumeration is a search through all possible combinations, controlled by the constraint solver that determines as soon as possible when current choices cannot lead to any (better) solution. The efficiency of implicit enumeration is heavily dependent on the heuristics used to order the search. A major asset of CLP with respect to combinatorial problem-solving is that it allows easy writing of complex dynamic heuristics based on a direct analysis of the initial problem data.

Research work at the Paris Scientific Center. The Paris Scientific Center is involved in the PROTOS project in the design and implementation of an extension of IBM Prolog by constraint solvers. Three main prototypes have been developed:

1. CLP(B) is an extension of IBM Prolog with Boolean constraints, developed in collaboration with Bang & Olufsen. It is very flexible and allows the user to choose different solving strategies, from a partial consistency check to a complete one that infers as many variable values as possible. Moreover, CLP(B) provides some high-level facilities including abduction, projection, and meta-programming with constraints. This prototype is efficient compared to other CLP languages and allows real-sized applications to be tackled.
2. A prototype of a solver for integer arithmetic has been completed and is currently being interfaced with IBM Prolog and PROTOS-L.
3. A solver for linear constraints on real variables is in the final phase of implementation.

Further work will be done to make these three solvers cooperate when solving complex problems.

Finally, research is being carried out along two lines:

- An algorithm is being designed to solve the disjunction of linear constraints on $\mathbb{R}$. It will incorporate some of our results about intelligent backtracking in CLP.[9,55]
- A general shell allowing scheduling problems to be easily described and solved is currently being designed, based on the above prototypes.

Types in Prolog. Although types play an important role in most modern programming languages, Prolog is essentially an untyped language. However, from a software engineering point of view, types can be vital in the development of reliable and correct software. Some type declarations make it possible to automatically reject meaningless expressions and terms (like `2 + paris`), provide a means for better structured programs, and make explicit the data structures used in a program. In PROTOS-L,[10] whose type concept is derived from TEL,[56] we could declare, for instance, `paris` to be of type `town` and `france` of type `country`. Further, the relation `in_country` could be said to take two arguments, the first of type `town` and the second of type `country`, as follows:

```
rel in_country: town × country
```

Thus, all uses of `in_country` are subject to type checking. If `in_country(paris,france)` would be accepted, for instance, `in_country(france,paris)` with swapped arguments would be rejected as being ill-typed, as would `in_country(42,france)`.

Furthermore, having such type and relation declarations, variables do not need to be declared. Instead, automatic type inferencing for variables is possible, allowing further programming errors

to be detected early at compilation time. For instance, given the relation declaration:

```
rel flight: flight_no × town × town
```

a clause containing:

```
. . . ,in_country(D,C), flight(FN, C, A), . . .
```

is rejected as being ill-typed because the variable C cannot be both of type country (as the second argument of in_country) and of type town (as the second argument of flight).

However, a term and thus also a variable may belong to more than one type if we allow for subtype relationships. For instance, the PROTOS-L type declaration

```
vehicle := airplane ++ car ++ train.
```

introduces the type vehicle as the union of its subtypes airplane, car, and train, where the type airplane could, for example, be given by enumerating its elements

```
airplane := {boeing747, dc10, airbus}.
```

The subtyping possibility greatly increases the representation facilities since the universe of discourse can now be subdivided and structured in a flexible way. Moreover, the deduction process can exploit the subtype relationships when testing for subtype membership or when restricting variables to subtypes. For instance, the travel relation declared by

```
rel travel: vehicle × town × town
```

could be defined with clauses such as:

```
travel(V,TownDep,TownArr) :- V:airplane, . . .
```

which is only applicable if the first argument is an airplane. When the incoming argument V is a variable, the subgoal V:airplane restricts it to the type airplane. If the previous type of V is incompatible with airplane, like car or train, it fails. (In fact, in this case the PROTOS-L indexing mechanism on typed variables would exclude this clause as an alternative to be considered right from the beginning.) Here, V represents the whole *set* of airplanes instead of a particular instance of this set. Thus, the deduction process uses the more abstract level of set-denoting types rather than the level of individuals. This yields not only more compact intensional answers, but it may also save a lot of expensive backtracking.

Often, one wants to express data structures in a parameterized way, and the most common parameterized data structures in Prolog are lists. Here is a declarative definition for appending two lists:

```
append([], L, L).
append([H|T], L, [H|TL]) :- append(T, L, TL).
```

which can be read as: (1) The empty list [] appended to some list L yields L, and (2) appending a list with head H and tail T to a list L yields the list with head H and tail TL, provided that appending T and L yields TL. (Note that this relational definition can be used both for appending two given lists, e.g., append([1,2,3], [4,5], L), and also for generating all possible splittings of a given list, e.g., append(L1,L2[1,2,3,4,5])). However, a problem with untyped Prolog is that a goal like

```
append([],2,2).                                    (*)
```

is also provable from this definition, in contrast to the intention that append is defined to operate on lists only. In untyped Prolog, we cannot express the applicability restriction. In the typed approach of PROTOS-L, the type of lists is defined by the *polymorphic* type definition

```
list(S) := {[], [_|_]: S × list(S)}.
```

Here, the variable S ranges over all types and can be substituted by any type description. Such a parametric definition makes available all list instances, e.g., list(vehicle), list(list(train)), or list(pair(train,time)) where pair(S1,S2) is another polymorphic type with two type arguments. Now, the goal (*) above is discovered ill-typed at compilation time as soon as append has been declared as follows:

```
rel append: list(S) × list(S) × list(S)
```

In order to support the type-checking facilities, currently every predicate in a PROTOS-L program must be declared. Among the advantages of this type system are those gained in traditional programming languages (like static consistency checks at compilation time, avoidance of mean-

ingless expressions, explicit data structures, and better structured programs). An additional advantage in logic programming is that computations on types can replace otherwise necessary deductions. This replacement may greatly increase efficiency by reducing backtracking. Thus, where it might seem cumbersome to give the typing information for each predicate in a small prototype, this information might be vital in a large application. Moreover, in PROTOS-L, compared to more traditional approaches, the typing effort is greatly reduced by the subtyping facility and the availability of both polymorphic types and predicates.

An open question currently under investigation is how typed program parts may be brought together with classical untyped ones (perhaps, by using a default "root" type for any untyped elements). Finally, in order to allow general meta-calls, the type system still has to be extended, e.g., in the direction of higher-order logic (see for instance Miller and Nadathur[57] or the recent work on Gödel[58] on meta-programming and the use of types).

Modules and abstract data types in Prolog. A PROTOS-L program is made of a set of modules. Each module consists of an *interface* and a *body*. The purpose of the module interface is to define the set of imported names and the names that are defined and exported by this module. The user of a module only sees its interface, not the body. In the following, we present only the two most significant abstraction possibilities enabled by this module system.

The first is the availability of abstract data types. In an interface we might have the declarations

```
interface planner.
    time_table := abstract.
    rel insert: meeting × time_table.
    rel cancel: meeting × time_table.
    rel free_time_slot: date × time
      × duration × time_table.
endinterface.
```

where abstract is a reserved word in PROTOS-L. Thus, the user of the module planner does not know the representation of the abstract type time_table, but may only use the exported relations like insert, cancel, etc. to access time_table. As a result, in the body of the module planner, the representation of time_table can be changed without having to change any other module. This abstraction mechanism corresponds to the *opaque* types in Modula-2 and the abstract types in TEL.

Modules and deductive database access. The second abstraction possibility enabled by the module system of PROTOS-L is the transparent access to external databases. The user of a module does not have to know whether an exported relation is implemented by a sequence of program clauses or by a relation in an external database. Consider the interface

```
interface products.
   rel needs: string × string × int.
            % product  product  amount
   rel depends_on: string × string.
                % product  product
endinterface.
```

which exports a relation needs(A,B,M) meaning that the product A needs the amount M of product B to be made, and a relation depends_on(A,B) meaning that the production of A depends on the product B via the relation needs, possibly involving intermediate products.

One possible implementation of these relations could be in an ordinary program body:

```
module products.
   rel needs: string × string × int.
       needs("product1", "product2", 50).
       needs("product1", "product3", 100).
       needs("product2", "product9", 80).
         . . .
   rel depends_on: string × string.
       depends_on(P1,P2) :- needs(P1,P2,A).
       depends_on(P1,P2) :- needs(P1,IM,A),
         depends_on(IM,P2).
endmodule.
```

However, another possibility is to state that the facts defining needs correspond to the tuples of a relation Product_needs in an external relational database Product_DB. This is achieved by the database body:

```
database_body products using Product_DB.
   rel needs: string × string × int.
       dbrel needs is Product_needs(Product,
         UsedProduct, Amount).
endmodule.
```

As for depends_on, it may have exactly the same definition as above, either in this database body or

in a separate program body. This simple example already shows the three levels of database access:

- *Base relations.* In order to access base relations in an external database one just has to state the correspondence between the PROTOS-L predicate and its arguments, and the database relation and its attributes. Here, the predicate `needs` corresponds to the database relation `Product_needs` whose attributes `Product`, `Used_Product`, and `Amount` give the first, second, and third argument of `needs`, respectively.
- *Views.* Using ordinary Prolog syntax, one can define database views by clauses in the same way a Prolog predicate is defined. (In fact, since there are no nested terms in relational databases, clauses in database bodies are necessarily function-free.)
- *Recursive views.* As is true with `depends_on` in the above example, a database predicate may be defined recursively, as easily as Prolog predicates. This goes beyond the power of SQL systems that do not allow recursion and plays a central role in deductive databases.

It is to be noted that the user can access an external database at any one of these three levels, without having to use a second language such as SQL. Moreover, PROTOS-L evaluates rules in a mixed bottom-up and top-down manner and reuses intermediate results.[59] On large relations (as typically are database relations), it is well known that this can result in a significant efficiency gain over the pure top-down Prolog evaluation strategy. The latter, however, allows complex terms to be used and may be faster on small relations. Thus, two different evaluation mechanisms are combined, each having its own merits. By choosing where to put, for instance, the recursive `depends_on` definition (in a program or database body), the user can choose the most appropriate evaluation method.

In the same way, access to the deductive database LILOG-DB[60] has also been integrated into PROTOS-L as a third kind of module bodies (`lilog_db_body`). Since LILOG-DB has a very powerful data model, including open and variant types, nested terms, and attribute-value notation, the relations realized in a `lilog_db_body` may contain arbitrary monomorphic and polymorphic types, whereas the interface to ordinary relational databases only supports integers, subtypes of integers, and strings.

In the context of database updates, PROTOS-L offers a transaction concept as the underlying database management system. For instance, update operations within a transaction are made permanent only if the transaction can be completed successfully; thus backtracking inside a transaction undoes every insert and delete operation.

Further extensions and prototype availability. Among further extensions of logic programming, PROTOS-L handles functions defined by conditional equations. It also provides various new built-in relations and functions related to types, e.g., for testing, instantiating, and generating typed variables. All built-ins are type-safe, including file input and output. Type-safe is also the interface to AIX*/Windows (Advanced Interactive Executive*/Windows) that was developed for the PROTOS system.[61] Through a collection of a few built-in predicates and types it provides an object-oriented access to the powerful window-handling facilities that have already been used extensively in the PROTOS-L applications. The implementation of the PROTOS-L system prototype is based on an extension of the Warren Abstract Machine to polymorphic order-sorted resolution, and it is currently available on the RISC System/6000 running AIX 3.1, on the RT/PC* 6150 running AIX 2.2.1, and on the PS/2 running AIX 1.2.

Scheduling applications: A practical application in Hoechst. In the planning area, PROTOS-L has been used successfully in various applications such as railway routing, map coloring, and production planning and scheduling (PPS) problems.[62] We concentrate here on a particular so-called single-step PPS application as it occurs in a fiber plant.

When entering the PROTOS project the Hoechst group had already built the EXAMPL planning system described in Jachemich.[63] It was decided to model this approach in PROTOS-L (HoPla system[64]) and to extend the work in the direction of further replanning facilities.

Motivation. Starting in the late 1960s, the fiber plant under discussion was (as many others) supposed to use a linear-programming-based system for its scheduling problems. Although the program worked and proposed an "optimal" solution, it turned out to be unsatisfactory with regard to several points. First, although believed to be optimal, the proposed schedule was almost incomprehensible. Second, the plant manager usu-

ally had several (conflicting) goal functions in mind that could not be simultaneously introduced into the system. Finally, the plant manager was not able to influence the program behavior other than through some very rough parameters, and inevitable "manual" changes to the proposed schedule usually led to more or less worse solutions. As a result, this program based on linear programming was only rarely used—most of the scheduling task was done with pencil and paper.

In general, it seems that purely mathematical solutions to PPS problems are unrealistic because of the following:

- Goals as well as parameters often are not easily converted into numerical descriptions.
- Mathematical solutions are hard to explain.
- "On the fly" changes to schedules are hardly manageable.
- If the environment changes, it is often hard to adapt such algorithms.

It was then decided to solve that scheduling problem using a knowledge-based system such that:

- The system should follow the same line of reasoning the plant manager uses when generating a schedule and should be able to explain its choices.
- "Manual" changes to the proposed schedule should always be possible, with the system incorporating these changes in an "optimal" way.

The problem environment. The fiber plant considered here is made up of several (up to 50) different single-step production lines. Raw material enters one end of a production line, and the finished product leaves the other end.

The problem is to schedule from 200 to 1000 orders within a specific period. Each order consists of a certain amount of some product that has to be delivered before a certain due date. The spectrum of possible products ranges over about 12000 different combinations of parameters such as (1) the basic fiber type, (2) the diameter, (3) the color, and (4) the bobbin type and size. A product may be manufactured more or less quickly on only some (up to 12) of the plant production lines. Finally, some products cannot be made simultaneously in the plant (because of incompatibilities between the dyes needed in the process).

When production proceeds from one product to another, the production line has to be reconfigured to some extent depending on the parameters of the preceding and following products. A major goal is to keep the overall reconfiguration effort as small as possible. The order due dates are a secondary criterion as long as all orders are scheduled within the considered period.

Implementation and replanning facilities. Instead of an algorithmic solution, the system models the plant manager's approach to scheduling: starting from an initial plan consisting of the orders that are already being produced or planned, the "best next order" on any of the production lines is searched for and inserted into the plan. What the "best next order" actually is, is determined by a set of heuristic rules such as:

- A best next order should have the same parameters as its predecessor.
- If one has to switch colors *or* diameters, take the same color and switch the diameter.
- If one has to switch colors, it is better to switch from a lighter color to a darker one.

These rules express the planners' knowledge on how to keep the resetting costs low. They can be translated directly into PROTOS-L, its typing mechanism providing a first level of consistency checking for the rule set.

The system offers an *explanation* facility. During the still ongoing computation, for every order that has been planned thus far, the user can already visualize its characteristics, the resetting costs it has caused, and, most of all, the reason why it has been inserted into this position in the plan.

Once the proposed schedule is complete, the user has a range of *replanning* facilities. In particular, the current plan may be changed by requesting that an order be allocated to a certain production line, or be the direct successor of another allocated order.

Instead of directly obeying the user's request, the system gathers all of the requested changes and regards them as new scheduling restrictions as soon as the user decides to start scheduling anew. This feature helps the plant manager avoid having schedules worsen each time some changes have to be made. Since the whole schedule can be redone, a "good" result will again be achieved.

However, if desired, the planner will be able to keep certain parts of the plan, e.g., the first week, or everything up to a particular order, on a specific production line.

Perspectives. Today, the scheduling application work in the PROTOS project is continuing in several directions. Whereas the scheduling problem described here takes place in a single plant and is thus an instance of local planning, distributed and global planning are now also being investigated.

The unique characteristics of logic programming, like high-level declarative programming and automatic search for alternatives via backtracking, make it an excellent choice for realizing knowledge-based systems such as the one described above. However, one should not blindly believe AI techniques to be the "one and only" solution to scheduling problems. A moderate mixture of AI and operations research approaches, as proposed, for instance, in the constraint logic programming paradigm, seems to be a feasible way. Moreover, only if AI parts are integrated in existing organizational software environments such as databases, will problems be successfully solved.

Conclusions

The last few years have seen tremendous growth in both the scope and depth of activities related to Prolog. Yet much is still to be done before Prolog is recognized for what it is.

First of all, the message that Prolog is a *general-purpose high-level programming language* has not yet been widely accepted. It is probably one of the best kept secrets that a lot of production applications are written in Prolog, and, in fact, the number is growing every year. In a recent announcement, IBM positioned Prolog in its AD/Cycle framework. That strategic announcement should certainly encourage its customers to use Prolog in commercial application development.

Second, there is a perception that Prolog is a difficult language to learn and master because it is based on mathematical logic, which may be intimidating. However, it is not necessary to know any logic theory to *use* Prolog efficiently. But it is thanks to this logic foundation that Prolog is unique and, contrary to most programming languages, has a solid theoretical framework.

Third, Prolog can be one of the keys to solving the long-standing productivity and quality problems of software development. The combination of logic programming, object-oriented programming, and constraint programming will provide a new dimension to the traditional way of developing computer applications, making it possible to address in a much more comprehensive way all of the declarative and descriptive aspects of computer applications, thus leading to significant improvements in productivity, be it in the development of the application or its maintenance. Many types of applications that seemed too difficult or expensive can become practical, ranging from traditional data processing applications through knowledge-based processing applications to natural-language processing applications.

Use of Prolog will lead to a change in the skills required and methods used, and will shift the emphasis from traditional programming and testing toward analysis or, in other words, from a computer solution to problem definition, with the benefit of an overall reduction in costs. Prolog mostly appeals to a specific class of programmers: typically those who are more highly skilled in conceptual and abstract reasoning, usually college graduates in computer science.

Finally, Prolog is particularly well prepared to exploit parallel architectures that are apt to prevail in the next computer generation.

Acknowledgments

We thank Wayne Chan, the Prolog product manager in the Paris Scientific Center, Ghislain Huyberechts, Bruno De Backer, IBM France, and J. Jachemich, Hoechst, Germany, as well as the anonymous referees for their valuable comments. We would also like to thank the following people who kindly and competently answered our many questions on the use of Prolog in their organizations: B. Botella, F. Desprez, P. Marguerie, P. Taillibert, S. Varennes, Dassault Electronique, France; F. E. Madison-Ferguson, Knowledge-Ware, Inc., USA; J. C. Miginiac, SOCS, France; A. Marien, L. Maes, J-L. Binot, BIM, Belgium; A. J. Symonds, P. A. Squitteri, IBM Mid-Hudson Valley Laboratory; R. Granat, D. Carney, IBM San Jose; E. Haller, IBM German Application Development Laboratory, Böblingen; R. Höltke, IBM GMTC Sindelfingen; E. Levy-Abbegnoli, P. Bertrand, IBM La Gaude Labora-

tory; T. Guillotin, IBM Paris Scientific Center; I. Bretan, IBM Nordic Laboratory, Sweden; A. Walker, IBM Research; P. Blot, Peugeot SA, France; and F. Masson, IBM IS, France.

In addition, we thank Rosalind Greenstein for her patience and apposite remarks in revising this paper.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of KnowledgeWare, Inc.

Cited references

1. J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the ACM* **12**, 23–41 (January 1965).
2. A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel, *Un Système de Communication Homme-Machine en Français*, Research Report, Groupe Intelligence Artificielle, Université Aix-Marseille II, France (1973).
3. R. Kowalski, "Predicate Logic as a Programming Language," *Proceedings IFIP 74 Congress*, Stockholm, North-Holland (1974), pp. 569–574.
4. D. H. Warren, *An Abstract Prolog Instruction Set*, Technical Report 309, Stanford Research Institute, Stanford, CA (1983).
5. B. Robinet, "Logic Programming at IBM: from the Lab to the Customer," *Proceedings of the Eighth International Conference on Logic Programming (ICLP'91)*, MIT Press, Cambridge, MA (1991), pp. 912–921.
6. P. Bellot, V. Jay, R. Legrand, and E. Perottet, "MILES, A New Step Toward the Integration of Logic and Functions," *Journées Françaises des Langages Applicatifs*, La Rochelle (January 1990).
7. A. Berard-Dugourd, J. Fargues, and M-C. Landau, "Natural Language Analysis Using Conceptual Graphs," *Proceedings of the International Computer Science Conference '88*, Hong-Kong (December 1988), pp. 265–272.
8. P. Dague, P. Devès, and O. Raiman, "Troubleshooting: When Modeling Is the Trouble," *6th National Conference on Artificial Intelligence*, Seattle (July 1987).
9. B. De Backer and H. Beringer, "Intelligent Backtracking for CLP Languages. An Application to CLP(ℝ)," *International Logic Programming Symposium*, San Diego (1991).
10. C. Beierle, "Types, Modules and Databases in the Logic Programming Language PROTOS-L," in *Sorts and Types for Artificial Intelligence, Lecture Notes in Artificial Intelligence, Vol. 418*, K. H. Bläsius, U. Hedtstück, and C.-R. Rollinger, Editors, Springer-Verlag, Berlin, Heidelberg, New York (1990).
11. O. Herzog and C-R. Rollinger, *Text Understanding in LILOG, Lecture Notes in Artificial Intelligence, Volume 546*, Springer-Verlag, Berlin, Heidelberg, New York (1991).
12. W. G. Wilson, "Prolog for Applications Programming," *IBM Systems Journal* **25**, No. 2, 190–206 (1986).
13. A. Walker, M. McCord, J. Sowa, and W. Wilson, *Knowledge Systems and Prolog: Developing Expert, Database and Natural Language Systems*, second edition, Addison-Wesley Publishing Co., Reading, MA (1990).
14. *Application Development Using IBM Prolog for OS/2*, GG24-3777, IBM Corporation (1992); available through IBM branch offices.
15. C. M. Yoder, *Applications for Prolog Series*, IBM Corporation, Poughkeepsie, NY (1992). (8 volumes.)
16. *IBM SAA AD/Cycle Prolog General Information*, GH19-6886, IBM Corporation (1992); available through IBM branch offices.
17. *IBM SAA AD/Cycle Prolog/2, Language Reference*, SH19-6888, IBM Corporation (1992); available through IBM branch offices.
18. *IBM SAA AD/Cycle Prolog/MVS & VM Language Reference*, SH19-6893, IBM Corporation (1992). (To appear.)
19. M. Benichou, P. Dague, J.-M. Gauthier, and J. P. Nigoul, "IBM Prolog Language Workbench," in *New Computing Techniques in Physics Research*, D. Perret-Gallix and W. Wojcik, Editors, Editions du CNRS (1990).
20. *IBM SAA AD/Cycle Prolog/MVS & VM Extension Feature Installation and Reference*, SH19-6897, IBM Corporation (1992). (To appear.)
21. J. Martin, *Information Engineering*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1990).
22. SOCS publication, *OCEANIC: Manuel de Référence* (1991).
23. A. J. Symonds, "Creating a Software-Engineering Knowledge Base," *IEEE Software* (March 1988).
24. P. Courtin, F. Doladille, and M. Muenier, "Diagnostic de Pannes dans un Satellite," *TECHNOSPACE—Journée IA et Espace (CNES)*, Bordeaux, France (December 1988).
25. P. Dague, P. Deves, P. Luciani, and P. Taillibert, "When Oscillators Stop Oscillating," *12th International Joint Conference on Artificial Intelligence (IJCAI91)*, Sydney (August 1991).
26. P. Deves, C. Fisher, and P. Taillibert, "Diagnostic à Base de Modèles: Une Alternative aux Systèmes Experts," *11ème Journées Internationales sur les Systèmes Experts et Leurs*, Avignon, France (May 1991).
27. P. Marrot and M. Muenier, "GIBUS: An Operational Expert System for Space Applications," *Applications of Artificial Intelligence VII Conference (SPIE/IEEE)*, Orlando (March 1989).
28. S. Florek and R. Höltke, "TADEX: Semiconductor Inline Test, Analysis & Disposition Expert Systems," *4th Workshop on Diagnostics and Classification*, Berlin (February 1991).
29. R. Höltke, "TADEX: Knowledge Maintenance Done by the Expert?," *IBM Expert Systems Newsletter*, Document Number ESN9102 (February 1991).
30. P. Bertrand, "La Conception de Circuits Intégrés Assistée par un Système Expert," *Les Utilisations Industrielles du Langage PROLOG, Afcet*, Paris (April 1990).
31. E. Lévy, "ESFA: An Extended Static Flow Analysis," *3rd Productivity and Process Tools Symposium*, Thornwood, NY (September 1989).
32. A. Berard-Dugourd, J. Fargues, M-C. Landau, and J-P. Rogala, "Natural Language Information Retrieval from French Texts," *Proceedings of the Third Annual Workshop on Conceptual Graphs*, St. Paul, MN (August 1988), pp. 3.1.3.1–3.1.3.4.
33. M-C. Landau, "Solving Ambiguities in the Semantic Representation of Texts," *Proceedings COLING-90* **2**, Helsinki (August 1990), pp. 239–244.
34. J. F. Sowa, *Conceptual Structures for Mind and Ma-*

chine, Addison-Wesley Publishing Co., Reading, MA (1984).
35. *IBM SAA Language Access General Information*, SH19-6680, IBM Corporation (1990); available through IBM branch offices.
36. G. Jonsson, "The Development of IBM SAA Language Access: An Experience Report," *Proceedings 7th International Conference on Data Engineering* (1991).
37. M. A. Sanamrad and I. Bretan, "IBM SAA Language Access: a Large-Scale Commercial Product Implemented in Prolog," *1st International Conference on the Practical Application of Prolog*, London (April 1992).
38. K. R. Apt, H. Blair, and A. Walker, "Towards a Theory of Declarative Knowledge" in *Foundations of Deductive Databases and Logic Programming*, J. Minker, Editor, Morgan Kaufman (1988), pp. 89–148.
39. N. Foo, A. Rao, A. Taylor, and A. Walker, "Deduced Relevant Types and Constructive Negation," *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seattle (1988), pp. 126–139.
40. D. Tzoar and A. Walker, *The Syllog Expert Database System: Notes for Users*, IBM Internal Technical Report, IBM T. J. Watson Research Center, Yorktown Heights, NY (1992). (Unpublished.)
41. A. Walker, "Backchain Iteration: Towards a Practical Inference Method Simple Enough to be Proved Terminating, Sound and Complete," *Journal of Automated Reasoning* (1992). (To appear.)
42. A. Beauvieux, "CONVENTION: A System to Organize Staff Travel," *3rd International Symposium on Artificial Intelligence*, Monterrey, N.L., Mexico (October 1990).
43. L. Maes, *Een Vertaler voor Twee Parallele Uitvoeringemodellen voor Prolog*, Ph.D. thesis, Department of Computer Science, K. U. Leuven, Leuven, Netherlands (May 1989). (A translater for two parallel executing models for Prolog.)
44. P. Weemeeuw, M. Bruynooghe, and M. De Hondt, "On Implementing Logic Programming Languages on a Dataflow Architecture," in *ESOP'88 2nd European Symposium on Programming*, Ganzinger, Editor (March 1988), pp. 359–372.
45. K. Ali and R. Karlsson, "Full Prolog and Scheduling OR-Parallelism in Muse," *International Journal of Parallel Programming* **19**, No. 6 (December 1990).
46. K. Ali and R. Karlsson, "The Muse Approach to OR-Parallel Prolog," *International Journal of Parallel Programming* **19**, No. 2 (April 1990).
47. H.-J. Appelrath, A. B. Cremers, and O. Herzog, *The Eureka Project PROTOS*, IBM Germany, Scientific Center, IKBS, Workshop, Zurich, April 1990, Stuttgart (1990).
48. J. Jaffar and S. Michaylov, "Methodology and Implementation of a CLP System," *4th International Conference on Logic Programming*, Melbourne, J-L. Lassez, Editor, MIT Press, Cambridge, MA (May 1987), pp. 196–218.
49. A. Colmerauer, "Solving Equations and Inequations on Finite and Infinite Trees," *Fifth Generation Computer Systems*, Tokyo (November 1984).
50. J. Jaffar and J-L. Lassez, "CLP Theory," *4th IEEE Symposium on Logic Programming*, San Francisco (September 1987).
51. A. Colmerauer, "Opening the Prolog III Universe," *BYTE* **12**, No. 9, 177–182 (August 1987).
52. N. Heintze, H. Jaffar, S. Michaylov, P. Stuckey, and R. Yap, *The CLP(ℝ) Programmer's Manual*, Monash University, Monash, Australia (June 1987).
53. P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, MIT Press, Cambridge, MA (1989).
54. K. Sakai and A. Aiba, *CAL: A Theoretical Background of Constraint Logic Programming and Its Applications*, ICOT, Tokyo (April 1988).
55. H. Beringer and B. De Backer, "Diagnosing Systems Modeled with Piecewise Linear Constraints," *Tools for Artificial Intelligence* (IEEE), Washington (1990).
56. G. Smolka, *TEL (Version 0.9), Report and User Manual*, SEKI-Report SR 87-17, University of Kaiserslautern, Kaiserslautern, Germany (1988).
57. D. Miller and G. Nadathur, "Higher-Order Logic Programming," in *Third International Conference on Logic Programming*, E. Shapiro, Editor, Springer-Verlag, Berlin (1986), pp. 448–462.
58. P. M. Hill and J. W. Lloyd, *The Gödel Report (Preliminary Version)*, TR-91-02, Dept. of Computer Science, University of Bristol, Bristol, UK (1991).
59. G. Meyer, *A Poor Man's Deductive Database, IWBS Report*, IBM Germany, Scientific Center, IKBS, Stuttgart (1992). (In preparation.)
60. T. Ludwig and B. Walter, "EFTA: A Database Algebra for Deductive Retrieval of Feature Terms," *Data & Knowledge Engineering* **6** (1990).
61. H. Jasper, "A Logic Based Programming Environment for Interactive Applications," *Proceedings 4th International Conference on Human-Computer Interaction*, North-Holland, Amsterdam (September 1991).
62. C. Beierle, "An Overview on Planning Applications in PROTOS-L," in *Proceedings 13th IMACS World Congress on Computation and Applied Mathematics*, R. Vichnevetsky and J. H. Miller, Editors, Dublin (July 1991).
63. J. Jachemich, "Rule Based Scheduling in a Fibre Plant," in *Proceedings 13th IMACS World Congress on Computation and Applied Mathematics*, R. Vichnevetsky and J. H. Miller, Editors, Dublin (July 1991).
64. H. Wittman, *An Example for Knowledge Based Production Planning with PROTOS-L*, Ph.D. thesis, University of Stuttgart and IBM Germany Scientific Center, IKBS, Stuttgart (1991). (In German.)

Accepted for publication August 4, 1992.

Michel Bénichou *Paris Scientific Center, Compagnie IBM France, 54 rue Roger Salengro, Péripole 115, 94126 Fontenay ss Bois, France (electronic mail: benichou@fribm11.bitnet).*
Mr. Bénichou is currently responsible for worldwide Prolog market development. He graduated from Paris Ecole Polytechnique in 1960 and joined IBM France in 1963. After two years in the Paris Service Bureau, he was in charge of French marketing and technical support in the mathematical programming area. In 1972 he participated in the creation of the IBM Paris Program Product Center where he was involved in the development of the first commercial integer programming code in IBM, MIP, a part of the MPSX program product. Then he participated in the design and development of MPSX/370 and MIP/370, the IBM Mathematical Programming program product. In 1980 he joined French professional services, spending several years in two large French banks helping them to redesign their information systems. In particular, he helped Indosuez Bank to design and develop OCAPI, a large software engineering integrated workbench. After specializing in expert systems customer support, he joined the Paris Scientific Center where he has worked in the Prolog devel-

opment team. He is the author of several international papers dealing with mathematical programming, software engineering, and Prolog.

Henri Beringer *CEMAP, Compagnie IBM France, 6876 quai de la Rapée, 75592 Paris Cedex 12, France (electronic mail: beringer@fribm11.bitnet).* Dr. Beringer graduated from Paris Ecole Polytechnique in 1984 and did his Ph.D. thesis work on natural language processing at the Paris VI University in 1988. During his thesis work, he designed and developed the CO-SYLOG language, an extension of Prolog for symbolic constraints. In Dassault Electronique, he first worked on natural language access to databases of technical documents. During 1988 he was at the Center of Research in Information Processing of Montreal (CRIM) where he was responsible for the MULTIQUEST project and carried out a generic interface to bibliographic databases using natural language. With IBM since December 1989, first in the Paris Scientific Center and recently in the CEMAP, he has been doing research on constraint-based diagnosis and on constraint logic programming. He is now leading the constraint part of the PROTOS project.

Jean-Michel Gauthier *Paris Scientific Center, Compagnie IBM France, 54 rue Roger Salengro, Péripole 115, 94126 Fontenay ss Bois, France.* A graduate of Paris Ecole Polytechnique in 1954, Mr. Gauthier joined IBM France in 1957. He first worked in the Paris Service Bureau on scientific applications (on the first IBM 704 computer installed in Europe). He then started a small operations research group and soon specialized in mathematical programming. In 1972 he participated in the creation of the IBM Paris Program Product Center where he was given the responsibility for the development of the first commercial integer programming code in IBM, MIP, and then MIP/370, feature of MPSX/370, the IBM Mathematical Programming program product. Then he became the product manager of MPSX/370. In 1980 he joined IBM France Professional Services. He spent several years in two large French banks helping them to redesign their information system; in particular, he was deeply involved in the development of a large integrated CASE workbench. After specializing in expert systems he joined the Paris Scientific Center to work in the Prolog development team. He is the author of several international papers on mathematical programming, software engineering, and Prolog.

Christoph Beierle *IBM Germany, Heidelberg Scientific Center, Institute for Knowledge-Based Systems, P.O. Box 80 08 80, D-7000 Stuttgart, Germany (electronic mail: beierle@ds0lilog.bitnet).* Dr. Beierle received a Diploma degree in computer science from the University of Bonn in 1980. Between 1981 and 1986 he was a research associate at the Universities of Bonn and Kaiserslautern, where he worked primarily in the area of formal foundations of software development and verification. In 1985, he received his Ph.D. from the University of Kaiserslautern with his thesis on algebraic implementation techniques. From 1986 to 1987, he held an IBM Postdoctoral Fellowship in the LILOG (Linguistic and Logic Methods) project of IBM Germany, doing work on knowledge representation formalisms used for natural language processing. Since 1988 he has been with the Institute for Knowledge Based Systems within the Scientific Center of IBM Germany. He is project leader for the international EUREKA project PROTOS (Logic Programming Tools for Building Expert Systems) in which advanced extensions of logic programming are developed and applied in knowledge-based planning applications. In 1991, Dr. Beierle received an IBM Outstanding Innovation Award for his work on the PROTOS-L system.

Reprint Order No. G321-5496.