Prolog for industrial software development

J. Vaucher[1], G. Bochmann[1], B. Lefebvre[1],
K. Lee[2], S. Vella[2], M. Wu[2]

(1) Département d'Informatique et de Recherche Opérationnelle,
Université de Montréal, CP 6128, Montréal, Qc, H3C 3J7
e-mail: Vaucher@IRO.UMontreal.CA

(2) IBM Canada Laboratory, 1150 Eglinton Ave. East, Station 2G, Dept. 570,
North York, Ontario, M3C 1H7
e-mail: KKLEE@torolab5.vnet.ibm.com

CANADA

Abstract

The paper describes the results of a project specifically designed to evaluate the utility of Prolog for industrial software production. The project was financed by the IBM Canada Laboratory and the work was done by the University of Montréal in collaboration with the Computer Research Institute of Montréal (CRIM).

Initially, the objective was to compare the industrial implementation of a telecommunications protocol with one developed in Prolog directly from the defining standard documents. The application chosen was an IBM software product in the last phases of development: MMS, the Manufacturing Messaging System of the Manufacturing Automation Protocol (MAP) - a protocol designed for the interconnection of robots, machine tools and computers in highly automated factories. It was hoped that the Prolog system could be designed as an executable description which could function both as a workable prototype and as a model for the validation and the testing of other MMS implementations.

Early in the project, another application of Prolog's particular strengths became apparent. Noting that about half of the manpower on project such as MMS was devoted to product testing, we concentrated on developing tools to improve productivity in this area and developed a suite of general tools which could be used, not only for the MMS project, but for other products as well. This new research direction proved more fruitful than the original plan and, since the end of the project, the Prolog tools developed on the MMS project have been used in production on other development projects.

The main conclusion of the MMS project was Prolog knowledge-based techniques are efficient and robust enough to be used for production in an industrial context.

The paper also comment on the particular strengths and weaknesses of Prolog that we observed during the project.

1. INTRODUCTION

The paper describes the results of a project specifically designed to evaluate the utility of Prolog for industrial software production. The project which lasted about one year from April 1989 to August 1990, was financed by the IBM Canada Laboratory and the work was done by the University of Montréal in collaboration with the Computer Research Institute of Montréal (CRIM). The University team was comprised of two full-time programmer analysts under the guidance of three professors; IBM assigned several persons to introduce us to the reality of commercial software development, to provide the required technical information and effect the technological transfer of our work back into the company. The internal reports describing the project are not readily available, but more information can be found in [Gamache, 1991, Vaucher, Bochmann, Desmarais, et al., 1991, Vaucher, Bochmann, Desmarais, et al., 1992].

Initially, the primary objective of the research project was to explore the implications of developing protocol implementations in Prolog. It was conjectured that the declarative high level nature of Prolog would enable software products to be developed directly from the protocol standard documents with a significant gain in either productivity or reliability. It was expected that a Prolog implementation would be less efficient than one in a language such as C; but, the project would provide concrete figures on the relative performance with respect to speed and memory. Finally, even if the Prolog implementation was too inefficient to be used in the field, it could serve in the validation of the standard or in the conformance testing of other implementations via the automatic analysis of test results.

To make things realistic, a software product in the last phases of development was chosen. This was MMS, the Manufacturing Messaging System of the Manufacturing Automation Protocol (MAP) - a protocol designed for the interconnection of robots, machine tools and computers in highly automated factories[GM Corp., 1988b, ISO, 1985, ISO, 1987b]. The researchers were asked to develop a workable prototype of the MMS system in Prolog. In what follows we will refer to our research activity as the MMSP project (MMS in Prolog) to distinguish it from the commercial implementation of MMS.

A Prolog implementation of the MAP-MMS protocol was developed. The code amounted to over 3000 lines code developed in Arity-Prolog on (and for OS2) by two people in 1600 man-hours. This program respected the general system architecture defined by the IBM designers, it could handle parallel users and it could communicate with the more traditional implementations. Execution was 10 times slower than the standard implementation.

Unfortunately, given the deadlines set for the project and in order to get an implementation that could be shown to match the industrial implementation, the Prolog coders followed the structure of the standard implementation. Furthermore, to help debugging and to achieve a modicum of speed, opportunities for backtracking were shunned in favour of deterministic code. As a result, the Prolog code had a structure very much like that of the C implementation. Secondly, with no-backtracking, it was no longer possible to run the program "backwards" as a checker or for conformance testing. Therefore, although this implementation fulfilled the stated research objective and showed that complex telecommunication software could be written in Prolog, it failed to demonstrate any convincing advantage for the use of Prolog.

Early in the project, it became apparent that there were other ways in which to use Prolog to improve the productivity of software development. The basic idea was that instead of

coding directly in Prolog, it would be more productive to use Prolog to write tools to generate the required code automatically.

Although this idea is quite attractive in theory, practice is another matter. For one thing, generators require input to define precisely what should be generated; it is not obvious what are the best primitives for such an input language so that it requires less effort to define what should be done than to write the code directly. Secondly, the design, implementation and testing of such generators requires considerable effort; it is not obvious that the additional effort will pay off.

The factor which tilted the balance in favor of the automated generators was the discovery amongst the specification documents of reliable exploitable knowledge. This information could be extracted automatically and then used to drive a large variety of different tools to assist in various areas of the project such as documentation, design and testing.

All large scale software projects start with complex and detailed specification documents. In the case of the MMS project, there existed a complete specification of the MMS user interface in the form of C definitions for all user accessible functions and data types[1]. As text, these definitions were moderately useful; but translated into Prolog clauses, the CDEF knowledge base (as it came to be called) could support many diverse applications. It allowed "browsing" and rapid linking of related information which was originally dispersed in many documents. It served as a basis for the interviewing front-end to a test-case generator. It also provided the information necessary to generate portions of programs automatically.

Due to the fact that the tools were not specific to the MMS protocol but based on a knowledge base of C specifications, it was relatively easy to transfer these tools to other projects and this is what happened after the contract was over.

In what follows, the academics give details of how they encoded and used the C binding information in a variety of tools. Then, the industrial sponsors of the project comment on the usefulness of these tools once they were put into practice. Finally, we conclude with some comments on the observed strengths and weaknesses of Prolog and the implementations that we used.

2. BUILDING THE KNOWLEDGE BASE

The relevant standard documents which we had available for MMS totalled over 2000 pages. At the beginning of the project, we tried parsing these documents in various ways, trying to extract exploitable information. After several attempts, we found that the C bindings of the service interface [GM Corp., 1988a, GM Corp., 1988b] were the most useful.

From a user's point of view, a package such as MMS can be considered to be a set of services accessed through a library of interface functions[GM Corp., 1988b]. MMS provides 103 basic services including file transfer, remote programme loading and execution as well as operations on distributed variables, events, and semaphores. For each such service, the standard provides a C prototype definition for the corresponding interface function. Also provided are C declarations for all basic types, data structures and constants that may be passed to or returned from these functions.

---

[1] For many standardised protocols, the data type information is available in the ASN.1 format [ISO, 1987a] and the techniques that we used could also be applied.

In addition to the primary service functions, another 40 or so functions are also specified to provide complementary operations such as the creation of environments (virtual machines) or the allocation and initialisation of the complex data structures required for the arguments of the other MMS functions.

Figure 1 gives a sample of the definitions concerning the function GET_NAME_LIST (This resembles the DIR function of the FTP protocol by returning the list of all entities accessible on a remote site).

```
Return_code mm_gnlist(      connection_id,
                        return_event_name,
                        input_dcb,
                        inout_dcb)

      Connection_id           connection_id;
      Local_event_name        return_event_name;
      Mm_GNList_in_dcb        *input_dcb;
      Mm_GNList_out_dcb       **inout_dcb;

      {  /* empty function body */  }


typedef struct Mm_GNList_in_dcb
{   Mask                  mask;
 #define          GNL_IN_DCB_CONTINUE_AFTER     BIT(0)
    Mm_Context_id         context_id;
    Mm_Object_class       object_class;
    Mm_Object_scope       object_scope;
    Mm_Identifier         domain_name;
    Mm_Identifier         continue_after;
} Mm_GNList_in_dcb;

typedef enum Mm_Object_scope
{
    Mm_OS_Vmd_specific,
    Mm_OS_Domain_specific,
    Mm_OS_AA_specific
} Mm_Object_scope;
```

Figure 1: Examples of "C" definitions from the Standard

To analyse the C bindings, we implemented a C definition parser in 400 lines of Prolog. For each definition, we generate a clause of the following form.

```
c_def( Name, Kind, Definition, Etc ).
```

which gives (1) the name of the defined entity, (2) its kind (i.e. constant, function or structure), (3) its definition (i.e. for a structure, the list of all field names and types) and (4) other related information such as where the definition was found. The only complex field is the *Definition* whose structure reflects the various forms of declarations possible in C. For the "mm_gnlist" function of Figure 1, the equivalent Prolog structure is

```
c_def( mm_gnlist,   function,
    [  'Return_code',
       [  connection_id = ['Connection_id'],
          return_event_name = ['Local_event_name'],
          input_dcb = ['Mm_GNList_in_dcb', ref],
          inout_dcb = ['Mm_GNList_out_dcb', ref, ref]]],
    ).
```

Analysis of all the definitions for the project takes 50 minutes on an IBM PS/2 model 80 with Arity Prolog. The complete data base has 2285 definitions and the text version occupies 250 Kbytes.

2.1. The Browser

One of the first tools grafted onto the CDEFs was a browser so that a developer could easily locate definitions and follow type links. Essentially, we added an index to the CDEF database so that any definitions could be located via the first *significant* letters of its name (independent of *case*). This enables the definition of **mm_read** to be located with the letters "REA" or the structure **Mm_GNList_in_dcb** to be found by giving "gnl". Possible choices are given in a menu and access is relatively fast (under 1 second with the full database). An example of interaction is shown below.

```
Search for:  read

  1  ...enough
  2  Mm_Read_in_dcb
  3  Mm_Read_info
  4  Mm_Read_out_dcb
  5  Mm_Read_specific_info
  6  mm_read

 Your choice:  6

mm_read: Function of type [Return_code]

Parameters:
   connection_id          = [Connection_id],
   return_event_name      = [Local_event_name],
   input_dcb              = [Mm_Read_in_dcb, ref],
   inout_dcb              = [Mm_Read_out_dcb, ref, ref]
```

The interface is relatively primitive with no graphics or scrolling menus because, at the time the system was developed, these features were not yet operational in our OS2 version of Prolog. This was a blessing in disguise since we were able to develop on a variety of machines including Macintosh, Sun and Apollo.

This browser was the first Prolog tool developed on the project. Its speed and immediate usefulness coupled with the fact that it was developed with relatively little effort was the first concrete indication to our industrial colleagues of what could be achieved with tools adapted to symbolic manipulation.

3. PRODUCTIVITY TOOLS FOR TESTING

When we joined the IBM project, the bulk of the implementation effort was over and the team was in the throes of testing. Our enquiries established that roughly half the man-power required on industrial development is devoted to product testing. Tools that could improve productivity in the area of testing would therefore have an important impact on the overall project performance. During our preliminary investigation of this area, our industrial colleagues stated that testing was too complex to be automated and that previous attempts to develop tools in the area had failed. On the other hand, our experience suggested that some of testing tasks could be implemented fairly easily in Prolog.

Given the situation: an economically important and useful task thought impossible to implement and a tight schedule, this seemed to be an ideal context to show how Prolog could best be used in an industrial context. Therefore, this was the area where we concentrated our efforts to design and implement tools that testers could and *would* use.

We do not pretend to have made an in depth study of test practice within either IBM or the industry; what we did was to automate some of the activities of the testers that we were in contact with. In what follows, we briefly describe what this testing entailed. Then we present the tools that we designed and implemented.

3.1. Testing in the context of MMS

As stated above, a protocol software such as MMS can be considered to be a set of services accessed through a library of interface functions. In our case, MMS provides 103 basic services each accessed by calling a separate interface procedure. In addition to these functions, additional utility functions are defined to provide such things as the allocation and initialisation of the complex data structures required for the arguments of the other MMS functions or the connection to a remote site. In all there are about 150 functions and procedures accessible to users of MMS.

The group in charge of testing MMS must check that each of these functions behaves correctly according to the specification documents. Moreover, since these functions are directly accessible by the users, the operation must be verified not only for reasonable values of the arguments but also for all possible values[2] including erroneous ones. In fact such thorough testing is limited through combinatorial explosion and the art of the testers lie in the design of small set of tests which (i) cover all typical parameter values and (ii) provoke at least once each possible error condition.

In practice, the testers that we observed started by determining, for each parameter field, a set of representative correct values and a set of representative erroneous values. Often, the tester would select values around the extremes of a permitted range. For example, given a parameter defined to be a positive integer (>= 0), the tester might choose {0, 1, **maxint**} as good values and { **minint**, -1 } as representative erroneous values.

Secondly, the tester plans a set of tests which cover these values. The basic objective of such a test set is that each correct and each erroneous value must be tried at least once. For incorrect values, there must be one test (call of the function under test) for each value; but, for good values, some economy is possible in that it is possible to test in parallel the

[2] For protocols, we should go even further and test behaviour in all possible states allowed by the protocol. In our work we limited ourselves to testing single function calls and did not consider exercising the states through sequences of service calls.

sets of good values for each parameter. An example will illustrate this. Assume that, for the function F(X,Y), the correct representative values are {a,b,c} for X and {m,n} for Y. All these values can be covered by just three tests: F(a,m), F(b,n) and F(c,m).

Although this technique is simple in principle, its application, in the case of a product like MMS is complicated by several factors:

1) Typically, the arguments which must be passed to each function are complex structures with many fields and sub-fields. Furthermore, many of these fields are pointers to list structures which must be created dynamically for each call.

2) Not only are there many fields to be considered, but there often exists interrelations between field values which must be respected. For example, for strings, there are often two fields: one which is a pointer to the string and the other is an integer which contains the length of the string. Another common case deals with variant records (or type unions) where the contents (or existence) of a given field depends on the value in another acting as a *tag*.

3) Finally, the situation is complicated by the fact that in a telecommunication environment, extensive initial and terminating actions are required to set up the context required by a test. For example, a second program may be required for our test program to communicate with. In addition, it may be required to set up a virtual machine (VMD), access a directory service, open a connection and use the values returned by these actions for the parameters of the function under test.

Therefore, an automatic testing tool must be able to compute test sets for large numbers of inter-related variables. It must also be coupled to a program generator which will tailor the code generation to the particular requirements of each test.

3.2 The testing support tools

The testing support system that we designed aims at testing single function calls. Given the specification of an MMS function, it computes sets of tests to cover the specified representative values and generates full test programs in C tailored to the function and the particular test values. The system is made up of three components:

1) The Interviewer
2) The test-case generator
3) The code generator

3.2.1 The interviewer

To test a function, one must first determine representative values (good and bad) for all parameter fields. Initially, this information is not in the CDEF database and it must be obtained from a human tester. To do this, the interviewer uses the parameter information stored in the CDEF database to guide a dialogue with the analyst prompting him with the name and type of each parameter field until all the following information has been obtained:

1) the input/output status (should a value be provided before calling a function or should the returned value be checked after ?),
2) a list of good representative values to be tried for the field,
3) a list of erroneous values
4) and finally a list of dependencies on the contents of other fields.

Once obtained, this information is added to the CDEF database and linked to the type definition of the field. In some cases, this can reduce the tester's work: if a type for which test information has already been entered is reused in another context, the old information is shown to the tester and he has the choice of keeping the old values or altering them. In the same vein, with enumerated data types, it is easy to generate and propose sensible representative values, but in all cases such proposals shown to the tester for approval.

The testing information for each parameter is kept as a pair of attributes with the following structure:

val = [*good values*] - *default_value* - [*erroneous values*],
cond = [link to other parameters] => *condition*,

The *condition* is a general sequence of Prolog goals and the values given in the "val" part are to be used only if this *condition* succeeds. To understand how these fields are used, consider the example below for **Mm_Identifier,** a type used quite often for strings in MMS parameter data structures.

```
known_type('Mm_Identifier',
   [pointer = ['Uint32',
      val  = ["Aa1_8","null"] - "null" - ["a 1"],
      cond = ( => true)],
   length = [ref - ['Octet'],
      type ='Octet_pointer',
      val  = [L] - L - [LL],
      cond = ([pointer = Pointer] =>
                     name(Pointer,Sy),
                     length(Sy,L), LL is L-1]
]).
```

The type is a structure with two fields: the first is a pointer to the string and the second gives the string's length. The definition gives "Aa1_8" and "null" as *good* representative values and "a 1" is a typically incorrect value (because of the space in the identifier). The length which depends on the string value used is assigned L and LL as good and bad values where L and LL are both obtained by computation from the value chosen for the *pointed* value.

3.2.2 The test-case generator

Once all parameter dependencies and representative values have been entered, the combination of values to be used for each test can be computed. The test-case generator uses one algorithm to generate good values and another to generate incorrect values.

For good values, the objective is to test all values with a minimum number of tests. Although it is possible to specify "combinatorial" groups of fields for which we want to generate all possible combinations of parameter values; mainly, the technique described above is used and sets of good values are explored in parallel. In the generation of error conditions, no such economy is possible: each erroneous value specified leads to a separate test with default values used for other parameters. In all cases, the specified dependencies are taken into account. Details of the algorithms can be found in [Gamache, 1991].

Variables	Representative Values	Test Number 1 2 3 4 5 6 7 8 9 10 11 12
connection_id	0	x x x x x x x x x x x x
return_event_name	0	x x x x x x x x x x x
	1	x
mask	bit(0)	x x
	bit(88)	x x x x x x x x x x x
continue_afterpoi...	"Aa1_8"	x
	"null"	x
continue_afterlength	5	x
	4	x
context_id	0	x x x x x x x x x x x x
object_class	Mm_OC_Domain	x
	Mm_OC_Event_action	x
	Mm_OC_Event_condi...	x
	Mm_OC_Event_enrol...	x
	Mm_OC_Journal	x
	Mm_OC_Named_type	x
	Mm_OC_Named_variable	x
	Mm_OC_Named_varia...	x
	Mm_OC_Operator_st...	x
	Mm_OC_Program_inv...	x
	Mm_OC_Scattered_a...	x
	Mm_OC_Semaphore	x
object_scope	Mm_OS_AA_specific	x x x x x x x x x x x
	Mm_OS_Domain_spec...	x
	Mm_OS_Vmd_specific	x
domain_namepointer	"Aa1_8"	x
	"null"	x x x x x x x x x x
domain_namelength	5	x
	4	x x x x x x x x x x

Figure 2: Valid test-cases generated for the **mm_gnlist** Function

Figure 2 shows the output of the generator for the valid test-case for the **mm_gnlist** function whose C definition was shown previously. The format of the output corresponds to that used by IBM testers to plan their work. Shown on the left are the abbreviated names of the INPUT fields. In the next column is shown the representative values for each field. The Xs on the right show which value is used for which test. The figure shows that 12 tests will be needed to cover *good* representative values. This number is determined by the 12 values defined for the "object_class" field. Note the *parallel* generation of the values for the different fields: in the first two tests, different values are generated, not only for "object_class", but also for "return_event_name", "continue_afterpoi...", and several others. The generated values are governed by several defined dependencies. In particular, the two "length" fields can be seen to have been

assigned the correct length of the corresponding strings. Finally, the two "continue_after..." fields exist only if the "mask" has the value "bit(0)" and the last two fields "domain_name..." exist when "object_scope" is equal to the value "Mm_OS_AA_specific".

The test-case generator also outputs its results in another form suitable for the next phase. The example shown below illustrates the form of the abstract syntax that we designed to specify function calls and parameter values:

```
/* ************** Case 1 *************** */

input = [
     connection_id = 1 ,
     return_event_name = 0 ,
     input_dcb .. ref .. mask = bit(0) ,
     input_dcb .. ref .. continue_after .. pointer = "Aa1_8" ,
     input_dcb .. ref .. continue_after .. length = 5 ,
     input_dcb .. ref .. context_id = 0 ,
     input_dcb .. ref .. object_class = Mm_OC_Domain ,
     input_dcb .. ref .. object_scope = Mm_OS_AA_specific ]
```

3.2.3 The code generator

This module can output complete C programs to perform the selected tests, including all initialisations, the dynamic creation and initialisation of the parameter data structures before calling the function under test and the checking of the returned results.

The generator uses several kinds of information to operate. First, there is the CDEF database which provides the basic information on the parameters for any function to be tested. Secondly, there is a library of rules and code patterns which reflect typical dependencies and sequences of code that we observed in programs written by the testers. This library takes care of such things as initialisation, declaration of variables and dynamic allocation of parameters. Finally, there is the actual description of which functions should be called with what parameter values.

For this project, we designed a simple notation with which to specify generated programs. The example below shows the use of this notation to specify a program (tsta0011) to test the MMS function **mm_status**. The program is specified as a sequence of function calls (just one here) and, for each call, attributes can be specified. The most important attributes (shown below) are (1) **input** with a list of assignments to input parameters, (2) **check** with a list of tests to be performed on the returned values

```
:- pgm(tsta0011,
     [ mm_status with [
          site = local,
          input=  [input_dcb..ref..context_id = 0,
                   input_dcb..ref..extended_derivation
                     = false],
          check= [inout_dcb..ref..[
                   size >= cast('Uint16')..0,
                   error_block..mask = '"allo"'],
                   input_dcb..ref..context_id = 0],
          output= [ inout_dcb = NULL] ] ]),
```

and (3) **output** with a list of variables to be unified with output variables (so that they could be used as input to subsequent function calls).

This specification caused the generation of a 170 line C program which was successfully compiled and executed. Parts of this program are shown in Figure 3 and the actual call of **mm_status** (shown in a box) can be seen to be only a minor part of the text.

```
/*****************************************
          mm_status
 *****************************************/

#include <stdio.h>
#include "mmssys.h"
   ...
void tsta0011 ()
{  ...
   Connection_id connection_id1 ;
   Local_event_name return_event_name1 ;
   Mm_Status_in_dcb * input_dcb1 ;
   Mm_Status_out_dcb ** inout_dcb1 ;
   ...
   rc = mm_didcb( MMiStatus ,ADD_SIZE,
                        &input_dcb, result) ;
   if (rc != SUCCESS)
     { print_error("mm_didcb",rc) ; return ; }
   ...
   input_dcb1  = ( Mm_Status_in_dcb * ) input_dcb ;
   ...
 local_connect( &loc_vmd_id,
                &loc_ae_id, &loc_connection_id);
   ...
   connection_id1 = loc_connection_id ;
   return_event_name1 = 0 ;
   input_dcb1->context_id = 0 ;
   input_dcb1->extended_derivation = false ;

      rc = mm_status(connection_id1,
                     return_event_name1,
                     input_dcb1,
inout_dcb1);

   if (rc != SUCCESS)
     printf("Error in function mm_status, rc=%d\n", rc);

   ERR_check(inout_dcb1->size >= ( Uint16 ) 0 ,
                "inout_dcb1->size >= (  Uint16 ) 0 ") ;
   ...
  rc = mm_dfdcb((Octet *) input_dcb1 , result) ;
   ...
   local_conclude( loc_vmd_id,
           loc_ae_id,
           loc_connection_id);
   return;
}
```

Figure 3: Selected portions of a generated program

The listing shows clearly the large number of elements which must be computed and generated (even in such a simple example), in particular:

- comments
- *include* statements for standard definitions
- variable declarations (variable names must also be created)
- memory allocation for dynamic parameters
- initialisation of the connection
- assignment to the parameters of the test values
- checking of returned values
- provision of error messages
- freeing of parameters
- disconnection

Figure 4 shows the input required for a more complex program. It includes 3 function calls, **mm_read** and **mm_leextract** (twice), direct invocation of generation statements (*out_dcl* and *out_code)* and the invocation of a recursive generation pattern, **make_list**. This example also illustrates how Prolog variables are used to link OUTPUT from one function generation with the INPUT to another (i.e. Lread output from **mm_read** used as input to **mm_leextract**).

4. EVALUATION BY THE INDUSTRIAL PARTNERS

After completion of the joint research project, the Prolog tools were tested on various development projects within IBM. These projects covered various areas such as communication protocols, distributed computing and databases, but they all had one characteristic in common with MMS: they provided services to applications via a library of functions collectively known as the Application Programming Interface (API). Furthermore, the interfaces to these functions were declared in C-language "include" files.

4.1 Applications of test tools

Experience indicates that different projects can make use of the tools in different stages. However, savings are realised mainly in the functional test stage which includes coverage and testcase generation. It is estimated that the test tools can save 70% of the manual effort in functional test which represents about 30% of all the test efforts. Hence, the test tools may save about 20% of overall test efforts.

Here are the testing activities where the tools were found most useful:

1. Binding analysis

The analyser, which converts C-language declarations into Prolog clauses, was found useful in the early stage of designing the C-language binding for an API. The browser could then be used to examine the knowledge in various ways. One example would be to list all types not referenced by any function. The analyser has successfully analysed over 10 different bindings with size up to 300K bytes and 6500 lines of C declarations.

```
pgm( trea001l,
[   out_dcl('char * data[10];') ,

    make_list('Mm_ET_Var_association',
          [um_nval("var1",'Mm_CL_Octet_string','data[0]'),
          um_nval("var2",'Mm_CL_Visible_string', 'data[2]') ],
              Assoc_List),

       mm_read with [
          input  =
              [input_dcb..ref..mask = 'BIT(1)',
               input_dcb..ref..context_id = 0,
               input_dcb..ref..var_assoc_list= *Assoc_List],
          output =
              [inout_dcb..ref..ref..list_of_access_result = Lread ] ],

       mm_leextract with [
          input  = [list =        Lread],
          output = [newlist  =  Lread2] ],

       mm_leextract with [
          input =    [list = * Lread2],
          output =  [newlist= Lread3] ] ,

       out_code( 'printf("data[0] = %s \n", data[0] ); ')
  ] ).

  make_list( _, [], L, L) --> [].
  make_list( Element_type, [ E_def | Es], List, FinalList) -->
      { E_def =.. [F |Args] , E_generator =.. [F, E1 | Args] },
      generate([
          E_generator ,
          mm_mlist with [
            input  =
                  [list = List,
                  element = cast('MMS_list_element')..(* E1 ),
                  element_type = Element_type],
             output =
                  [newlist= NewList] ],
       make_list( Element_type,  Es, NewList, FinalList)
  ] ).
```

Figure 4: Specification of a more complex program

2. Coverage generation

The interviewer guides the tester in entering representative test values for each parameter and the testcase generator then optimizes the combination of test values and generates the test coverage both in Prolog and table form. Even if code generation is not included into the test strategy, this coverage generation feature was found useful for functional test purposes and the testers often used the printed table to create testcases by other means.

3. Testcase code generation

As described previously, the code generator can be used to generate the entire testcase for simple functions, including comments, declarations, memory allocations, assignments, etc... However, for more complicated cases, the testers often proceeded in a slightly different way. They used the code generator to produce modules which handle just the assignment of test values and these modules were then linked with other test programs to perform more complex operations as stress or performance testing. The main advantage of using the tools is that once the test coverage is reviewed, the testcase generation part is automatic.

4. Output code generation

Most, if not all, testing or debugging programmes require output routines which can print out the contents of complex data structures in a readable way. This means indenting the output, printing out the name of each field as well as its contents. Furthermore, in the case of "enumerated" datatypes, this means printing out the symbolic value, not the integer code, for example: printing out TRUE and FALSE instead of "1" and "0". The structure of the code for these output routines is quite simple and repetitive; but the writing of all the routines required by the test programmes for a new application is a tedious and time-consuming job.

A user of our tools noted the similarity between this task and the testcase code generation that we had done and he implemented an extension to the tool suite to automate the generation of these output routines from the knowledge base. This is an interesting development since it was done by a "user" distinct from the original project team.

4.2 Limitation of the Prolog tools

Although the test tools proved useful in certain test applications, there are still obstacles to their wider acceptance. First, it is evident that more development work is necessary before these experimental prototypes can be accepted into standard practice. In particular, the user interface (that we purposefully avoided implementing) needs improvement. Furthermore, testcase generation is just part of a tester's job and more work would be required to provide a complete integrated test environment.

A different kind of problem is the lack of Prolog experience of the current breed of programmers. The power of Prolog comes from interactive use and incremental modification by users knowledgeable both in Prolog and in AI or meta-programming techniques. In particular, users have to learn Prolog if they want to customize our tools for their application; and, Prolog being so different from conventional languages, this entails expense and effort which was not usually foreseen in testing budgets.

4.3 Industrial conclusions

The most important feature in the test support tools is the conversion of a C binding into a Prolog knowledge-base: a knowledge-base which can be used not only in testing but also in other areas such as application generation. Other usage of the tools will be explored in the future.

Secondly, our experience shows that Prolog is very suitable for the design of knowledge-based systems such as the test support tools mainly because of its flexibility in knowledge representation. The effort required to implement the same system with a rigidly typed languages such as C or Fortran would have been prohibitive.

5. PROLOG STRENGTHS

Key to the success of our tools was the exploitation of Prolog's unique symbolic manipulation capabilities. Two other traits were also important: portability and interactivity.

5.1 Symbolic computation

In Prolog, it is easy to read, manipulate, organize and compare symbols. Furthermore, list and trees are integrated into the language and variables obviate the need for explicit allocation and freeing of memory.

The parsing of the C definitions as well as the code generation made use of the built-in Definite Clause Grammars (DCGs). For the CDEF database, Prolog functioned as a relational database; better even, since most database systems can't handle arbitrary tree structures and variable length strings. The browser relied on unification for pattern matching. For the interviewer and the code generator, we turned to meta-programming. For all these types of computation, traditional algorithmic languages such as C, FORTRAN or PL/1 would have been highly inappropriate.

5.2 Portability

Although the applications were delivered in Arity Prolog on a PS2 platform, development work was also done on other machines. For example, most of the tools described here were first developed on Macintosh using AAIS Prolog. We also used Quintus Prolog and C-Prolog running on Suns and Apollos. The ease with which code could be ported between machines meant that we were able to reuse many predicate libraries that we had previously developed in other contexts.

5.3 Interactivity

The ability to intermix interpreted and compiled code is also important. Interpretation is much better for exploratory prototyping and debugging whereas compilation is useful for performance.

6. PROLOG WEAKNESSES

Despite its many advantages, Prolog was not designed for large scale industrial work and it lacks some "standard" programming features that are useful for reading, maintaining and debugging large programs. In the MMSP project, we noted the following problems:

- lack of "records" to access structure fields by name
- lack of standardised types
- weaknesses in the support of large data
- lack of Prolog databases

6.1 Record structure

Prolog represents structured data types (arrays and records) as trees and implements access to fields via pattern matching. This leads to cryptic code which is difficult to understand, debug or maintain. At the University of Montréal, we are quite aware of these deficiencies and have developed several extensions which add "structure" to basic

Prolog. One such extension is a "record" feature which allows access by name to the fields of complex data structures [Vaucher, 1989].

Given the complexity of the data structures involved in the MMS specifications, it was obvious from the start that such a "record" feature would be essential to the success of the project. However, the "record" package had never been tested on a project of this magnitude and early experimentation with MMS pointed out some shortcomings. This led to the introduction of some new operators (**with** and **with...gives**) to simplify the access notation. Furthermore, we extended the definition capabilities to include hierarchies of records and typed fields but the basic access mechanism described in [Vaucher, 1989] has remained unchanged. The examples below give a taste of the improved "record" feature. As with a Pascal types, a **record** (and its fields) must be declared before it is used. Here we show the declaration of an "mm_status" record type with four *typed* fields along with the declaration of one of these field types:

```
def_record   mm_status(
     connection_id          : 'Connection_id',
     return_event_name      : 'Local_event_name',
     input_dcb              : 'Mm_Status_in_dcb',
     inout_dcb              : 'Mm_Status_out_dcb').

def_record 'Mm_status_in_dcb'(
     context_id             : 'Mm_Context_id',
     extended_derivation    : 'Bool').
```

Once a record type has been declared, instances can be created and fields accessed by name without the need to know exactly the structure of the record. Thus also means that record structures can be modified without altering the text of already-written code. The example below shows the retrieval of an instance of "mm_status" with "extended_derivation" equal to **true** and the setting of a subfield of the "input_dcb" to 111.

```
P instof mm_status, P,
P with [  input_dcb..extended_derivation = true,
          input_dcb..context_id := 111 ], ..
```

The extended "record" facility has proved robust, efficient and practical. It was used in all our software and it is doubtful that we could have handled reliably the complex data structures of MMS without it.

6.2 Basic types

When porting Prolog code, we found that it was the little things which were handled differently and gave most of the problems. In particular, we had problems with strings and characters. In AAIS Prolog[Lanam, 1988], these are first class types and there is special provision for their input and output and tracing. This is not the case in other Prologs: for example, in Quintus-Prolog strings are translated into lists of integers and become unreadable in subsequent traces. Similarly, there are differences in the notation for special characters such as single (') and double quotes ("). It would be useful if both characters and strings were adopted as standard types and treated uniformly by all implementations.

6.3 Weaknesses in the support of large data

During the project, we found two implementation "bugs" which denoted weaknesses in the handling of large data structures. The first dealt with long lists. Although there is no apparent limit to the number of elements in a list kept in internal storage, attempts to *read* long lists caused problems in both Arity and AAIS Prolog. The solution was to split external lists into smaller chunks and recombine them once read.

Another problem was that the atom space (for the symbol table) for the Arity implementation of Prolog is limited to a 64K space. This is too small to handle the number of distinct symbols used in our C bindings. This limitation can be avoided by using "strings" instead of symbols but this slows and complicates both inference and pattern matching. Finally, we implemented a "cache" mechanism which allowed strings to be used in the bulky knowledge base and converted strings to symbols automatically when required for processing.

6.4 Support for Databases

In spite of appearances, Prolog's support of large databases is flawed. Some Prolog implementations provide an interface to standard database systems but the data that can be stored there is subject to the limitations of classical databases and the storage of Prolog's dynamically-created arbitrary graph structures is excluded.

Secondly, although Prolog clauses are often seen to operate as relations, in fact, there are often limits (such as pointed out in the previous section) on the size of data that can be effectively handled. Furthermore, data stored as clauses in a program can't be shared between several applications (or protected) as one would expect from a database.

What is required is a storage system for reliable shared access to large-scale persistent data of the type found in Prolog (arbitrary trees with variable length data and variables). There are now object-oriented databases; what is needed are **Prolog-oriented** databases.

7. CONCLUSIONS

The major conclusion of the MMSP project is that:

> Prolog knowledge-based techniques are efficient and robust enough to be applied in industrial-scale applications.

Although we knew that Prolog was theoretically suitable for the kind of symbolic manipulation that we proposed to do, it remained to be proved whether it could handle the vast amounts of data present in a real industrial application. Although we did find some faults in the particular implementation of Prolog used for the project; we found that all the knowledge useful for the application (the Prolog C bindings definitions totalled 250 Kbytes in text form) could be stored and accessed rapidly.

However, Prolog should be used selectively in areas where its unique symbolic manipulation capabilities are required. In this project, Prolog seemed mainly useful in the form of knowledge intensive tools: browsers, specification (and C) analyzers, test sequence generators, test coverage computation, program generators, etc... Although we were able to program parts of the MMS protocol itself in Prolog, the Prolog version was painfully slow and there seemed to be little advantage in implementing this kind of real-time application in Prolog.

The industrial partners for the project confirm that the knowledge-based tools developed during the project, even in their rough experimental state, were employed (albeit in a limited way) and found to be useful on other projects. In one case, the code generation capabilities were extended in a totally new direction. It is estimated that use of the test tools could save about 20% of overall test efforts. However, more development work is necessary before these prototypes can be accepted into standard practice. In particular, the user interface needs improvement. Another problem is that the power of Prolog comes from interactive use and incremental modification by users knowledgeable both in Prolog and in AI or meta-programming techniques. This kind of training is not yet widespread.

Finally, we pointed out several areas pertinent to industrial work where current Prolog implementations could be improved. In particular, we mentioned: "record", lexical standards for basic types and the need for databases specifically adapted to the requirements of logic programming.

ACKNOWLEDGMENTS

The project was financed by the IBM Canada Laboratory in Toronto and the work was done at the University of Montréal in collaboration with the Computer Research Institute of Montréal (CRIM). Through its operating grant 7699, the Natural Sciences and Engineering Research Council of Canada also helped; specifically, the Council financed the ongoing work before and after the contract period including this presentation.

Most of the coding for MMSP was done by P. Gamache and S. Desmarais of the University. J. Lebensold of CRIM served as project administrator and J. Labatt provided ongoing inspiration. We also want to mention all the IBM personnel who took part in the project: in particular, Gordon Lee who started it and Tony Rego who kept it alive. We are also thankful to Paul Cheung, Joanne Downey, Alvin Fung, Payman Hodare et Simon Ng who initiated us into the mysteries of MMS.

REFERENCES

1. Gamache, P. "Générateur intelligent de tests adaptés au domaine des protocoles de communication." MSc Département d'informatique et de RO, Université de Montréal, CANADA, 1991.

2. GM Corp. "Application Interface Model and Specification Requirements: MAP_Attachement 1 to MAP_Appendix 7." Map-A7A1 N° General Motors Corporation, June 1988a.

3. GM Corp. "MMS Application Interface Specification (API): MAP_Attachement 6 to MAP_Appendix 7:." N° Map-A7A6, General Motors Corporation, June 1988b.

4. ISO. "Manufacturing Message Specification - Service Definition." Draft Intern'l Standard N° ISO/DIS 9506/1, ISO/TC 184, 1985.

5. ISO. "Information Processing - Open systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)." N° IS 8824, International Standards Organisation, May 1987a.

6. ISO. "Manufacturing Message Specification - Protocol Specification." Draft Intn'l Standard N° ISO/DIS 9506/2, ISO Technical Committee, December 1987b.

7. Lanam, D.H. "Advanced A.I. Systems' Prolog: Reference Manual." N° Version M-2.0, Advanced A.I. Systems (AAIS), Inc., PO Box 39-0360, Mountain View, California, 1988.

8. Vaucher, J. "A Record Package for Prolog." Soft. Pract. & Exp., 19 (8), p. 801-807, 1989.

9. Vaucher, J., G.v. Bochmann, S. Desmarais, P. Gamache, et al. "Le projet MMS: l'intelligence artificielle appliquée à l'implantation et au test de logiciels industriels." Publication N° 791, Dep. d'informatique et RO, Université de Montréal, Août 1991.

10. Vaucher, J., G.v. Bochmann, S. Desmarais, P. Gamache, et al. "Le projet MMS: l'intelligence artificielle appliquée à l'implantation et au test de logiciels industriels." Intelligence Artificielle et Sciences Cognitives au Québec, 3 a paraître 1992.