

## PROTEL: A HIGH LEVEL LANGUAGE FOR TELEPHONY

D.G. Foxall, M.L. Joliat, R.F. Kamel, J.J. Miceli

Bell-Northern Research  
Ottawa, Canada

### Abstract

PROTEL is a state-of-the-art high-level programming language designed specifically for telephony applications. It is widely used at Bell-Northern Research in the development of modular software for the DMS-100<sup>1</sup> series of digital switches. This paper describes the major features of the PROTEL language and its support system.

### 1. Introduction

The PROcedure Oriented Type Enforcing Language (PROTEL) is a high-level programming language designed at Bell-Northern Research for the development of software for large digital switching systems. The need for PROTEL arose as part of the evolution of the DMS-100<sup>1</sup> series of telephone switches - a series that combines the proven versatility of stored-program control with the numerous advantages of digital switching [1,2].

Digital telephony imposes a number of requirements not shared by other computer systems. Reliable uninterrupted service must be provided by a telephone exchange 24 hours a day, 7 days a week, even when the switch is being reconfigured. Operational efficiency is dictated by real-time constraints: the system must respond within predetermined time limits. In addition to meeting these requirements, BNR's objective in designing DMS-100 is to provide a digital switch general enough to cover the needs of local, toll, gateway, and combined telephone office applications. The software must be flexible enough to meet future requirements, and adaptable enough to suit different office needs without major modification. Ease of maintenance is equally important. Thus the objectives of the DMS-100 series are reliability, efficiency, generality, adaptability, maintainability, and especially modularity.

The key objective of DMS-100 software is modularity. Programs should consist of a number of autonomous modules with narrow well-defined interfaces. This encourages structured system design in which a problem is subdivided into a number of modules, the interface of each module is specified, and modules are independently implemented and tested.

Modularity also helps realize some of the other DMS-100 objectives as described above. For example, since the scope of changes is often

localized to a module, the maintainability of the software is enhanced. It is our contention that explicit modularity is a concept of fundamental importance in large scale software design and implementation. Consequently, modularity must be enforced in DMS-100 development tools. The most important such tool is the implementation programming language.

It was decided that a high-level language should be used, in keeping with the trend in the development of reliable software. The language should encourage structured programming and enforce strong type checking. The most promising candidate for DMS-100 implementation was PASCAL [3]. However, the facilities in PASCAL for the support of modularity were lacking. In particular, PASCAL did not allow a separate compilation mechanism. Other shortcomings of PASCAL, in the context of DMS-100, were the lack of an orthogonal set of data structures (eg - arrays of structures were not easily supported), the lack of variable-length arrays, etc. [4,5]. Although it is possible to enhance PASCAL to meet the needs of DMS-100, it was felt that the required modifications were too extensive. Thus the decision was made to design and implement PROTEL. Furthermore, it was decided that PROTEL should be the only implementation language - no assembly language was provided for DMS-100.

Thus, the major design goal of PROTEL is to enforce and support explicit modularity. The language encourages a hierarchical system structure in which higher level modules (eg - call processing) use lower level modules (eg - the operating system). An offshoot of this goal is the requirement of separate compilation. Other design aims met by PROTEL are extensive type checking across separate compilations, high-level control constructs, an orthogonal set of powerful data structures including dynamic arrays, generation of efficient code, and programmer control over emitted code sequences.

### 2. The Language

PROTEL is a block-structured, strongly-typed language that draws upon the features of PASCAL [3], MARY [6], and ALGOL 68 [7]. This section presents a synopsis of PROTEL using interrelated examples.

A PROTEL program consists of a hierarchy of interconnected modules, which in turn are constructed from sections. A section, the basic unit of compilation, is composed of a sequence of global

declarations, some of which may be procedure declarations. Procedures may contain local declarations and executable statements.

**2.1 Type Declarations.** PROTEL exploits the demonstrated advantages of extensive type specification [3]. These advantages include enhanced readability, reliability, and maintainability of programs.

Type declarations associate an identifier with a set of data attributes. These attributes may be described in terms of primitive language types or in terms of previously defined type identifiers. The primitive types are divided into simple types, aggregate types, and procedure types.

Simple types describe single data items. These are numeric and symbolic ranges, booleans, and pointers. For example:

```
TYPE digit_value, terminal_id {0 TO 9}
    status_condition {busy, idle, blocked, ready},
    out_of_service BOOL,
    protocol_ptr PTR TO status_condition;
```

The numeric type `digit_value` specifies a closed integer subrange. Instances of type `status_condition` may assume one of the values in the symbolic range. Boolean values may be either TRUE or FALSE. Items of type `protocol_ptr` are restricted to point at objects of type `status_condition`.

Aggregate types define collections of items. These include sets, tables, structures, and the special PROTEL types - descriptors and areas - neither of which is supported in PASCAL. For example:

```
TYPE digit_register TABLE {0 TO 19} OF digit_value,
    special_feature SET {abbreviated_dial, add_on,
        call_transfer, do_not_disturb},
    time_interval
    STRUCT
        amount {0 TO 255},
        unit {ten_ms, secs, mins, hours}
    ENDSTRUCT;
```

`Digit_register` specifies a 20 element array, whose elements are of type `digit_value` (defined previously). Instances of the `special_feature` type may assume any of the possible  $2^4$  subset values. The `time_interval` structure defines a type that consists of two elements: a numeric amount and a symbolic unit of measurement.

Descriptors are used to indirectly reference tables, and can be explicitly manipulated to provide support for references to run time allocated (dynamic) arrays. Run time allocation avoids the need for recompilation when changes in program requirements force data tables to be modified. This is especially important in a telephony environment where call processing data tables vary in size from office to office.

A descriptor consists of two parts: a descriptor part and a table part. The descriptor part con-

tains the table element count, the size of these elements, and a pointer to the base of the table part. The table part contains the elements of the current array. At execution time a descriptor may be made to point at any table of the correct type. This results in modification of the element count and pointer fields of the descriptor part. For example:

```
TYPE time_table DESC OF time_interval;
```

defines `time_table` as a descriptor which at run time points to a table whose elements are structures of type `time_interval` defined above. Actual storage for this array may be allocated statically at compile time, or obtained at run time by a call to a system storage allocator.

The area type provides the ability to define structures in which the declaration of elements may be postponed or distributed, and thus hidden. The principle of data hiding is important in the implementation of large scale modular software, such as in DMS-100. An area may be either a father area or a refinement of a father area. A father declaration indicates the amount of storage required for the area, and may also specify some of the elements within it. A refinement specifies the names and types of other items in the area. Several refinements may be associated with the same father. In such cases, the refinements are overlaid in storage. For example:

```
TYPE daddy AREA (6 * byte_width)
    i integer
ENDAREA;

TYPE son AREA REFINES daddy
    t TABLE {0 TO 15} OF BOOL
ENDAREA;

TYPE daughter AREA REFINES daddy
    j integer,
    k BOOL
ENDAREA;
```

The area `daddy` contains 6 bytes. The first element is an integer item `i`. Son and daughter refine `daddy`. The element of son overlays the elements of daughter following integer `i` in storage.

PROTEL procedures are data types, and as such may be used in type declarations. They may also be used in the definition of procedure variables and constants. Neither procedure types nor procedure variables are fully supported in PASCAL. A procedure type declaration describes formal parameters, their attributes, and optionally the attributes of a return value. The parameter attributes include the parameter type and its passing mode: by value (the default), by (read-only) reference, or by updates. For example:

```
TYPE get_channel PROC (terminal terminal_id)
    RETURNS integer;
TYPE transfer PROC (REF t time_interval,
    UPDATES feature_table DESC OF BOOL);
```

A typical use of procedure variables in DMS-100 software is the dynamic selection of procedures at run time. For example, a set of I/O procedures (eg - open, close, input, output) may be written for each device in a system. The correct device handler is selected at run time by indexing a table of procedure variables with the device type. New devices are introduced by adding their corresponding I/O procedures to the system, and modifying the procedure variable table. Such additions may be performed without changing the base system software. The same effect may be achieved using CASE statements, however all such selections must be modified and the corresponding code recompiled when new devices are added. Thus the use of procedure variables provides improved modularity, flexibility, and adaptability in DMS-100.

**2.2 Data Declarations.** Data declarations associate type attributes with a constant or variable. They also specify values for constants or optional initialization values for variables. For example:

```
DCL remote make_busy BOOL;
DCL inter_digit_time interval IS {15,secs},
   line_status status_condition INIT {idle},
   matrix TABLE {0 TO 100} OF digit_register,
   digit_buffer DESC OF digit_value;
```

Since procedures are types, data declarations may be used to declare procedure constants and variables. A procedure constant declaration defines the procedure body as the value of the constant. A procedure variable may be bound at run time to a specific procedure constant. For example:

```
DCL proc_var get_channel
DCL proc_const get_channel IS
   BLOCK
      proc_const -> proc_var; % assignment
   ENDBLOCK;
```

**2.3 Data References and Operations.** Data items are normally referenced by their names, however elements of data aggregates require further specification. Single table or descriptor elements are qualified by an index; a contiguous group of elements is accessed by an index subrange called a "slice". Structure and area elements are referenced by qualifying the name of the aggregate with the name of the element. The following data references use previous declarations:

```
inter_digit_time
digit_buffer [5 TO 10]
matrix[i][j]
inter_digit_time.unit
```

The first example refers to the entire structure `inter_digit_time`. The index subrange 5 to 10 refers to the slice consisting of the fifth through the tenth elements of `digit_buffer`. `Matrix[i][j]` refers to the *i*-th element of `matrix`, which is itself a complete table. The last example specifies the unit field of the structure `inter_digit_time`.

PROTEL supports the usual arithmetic, comparison,

and logical operators. It is noteworthy that assignment (`=`) is an operator and is treated like any other binary operator. PROTEL expressions are evaluated from left to right with no implicit operator precedence. This was done to avoid arbitrary decisions concerning the precedence of uncommon operators (eg - logical shift, exclusive or) which are implemented in PROTEL. The order of evaluation may be modified by using parentheses. For example:

```
2 + 3 * 7 -> i = 23 -> target;
```

The above expression evaluates to FALSE. The value assigned to *i* is 35. This yields the same results as the fully parenthesized expression

```
((((2 + 3) * 7) -> i) = 23) -> target;
```

**2.4 Control Structures.** PROTEL provides a variety of control structures to modify the normal sequential execution of statements. Conditional execution is accomplished by the IF..THEN..ELSE..ENDIF construct. The CASE and SELECT statements provide a selective execution facility. Both constructs perform the same function, but the former is optimized for execution speed, while the latter is optimized for storage. In some applications, it is acceptable for a compiler to deduce the optimal form of emitted CASE or SELECT code. However in a real-time application such as DMS it is important that the programmer be allowed to choose the most suitable construct explicitly.

By bracketing a group of statements with the DO...ENDDO pair, an infinite loop is obtained. Iteration is controlled by associating FOR and/or WHILE modifiers with the DO statement. PROTEL does not have a go-to statement; however, an EXIT statement provides a controlled escape mechanism. For example:

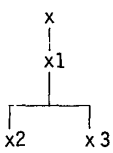
```
1) DCL temp digit_value, result integer,
   terminal terminal_id;
   % assign the proc constant to a proc variable.
   proc_const -> proc_var;
   IF remote_make_busy & (line_status = idle)
      THEN proc_var (terminal) -> result;
   ELSE
      matrix[3][10] -> temp;
   ENDF;
2) DCL not_found BOOL INIT TRUE,
   is_valid PROC(index {0 TO 19}) RETURNS BOOL
   IS FORWARD;

   loop:
   FOR i FROM 19 DOWN TO 0 WHILE is_valid(i)
   DO %halt loop when zero is found.
      IF (matrix[i][i] = 0 -> not_found) THEN
         EXIT loop;
      ENDF;
   ENDDO;
3) CASE first_digit_dialled IN
   {0}: TRUE -> operator;
   {1}: TRUE -> ddd call;
   {4}: TRUE -> assist;
        check 11();
   {2,6}: TRUE -> wrong_number;
   OUT % handle all other digits
        TRUE -> local_call;
   ENDCASE;
```

**2.5 Program Organization.** As discussed previously, PROTEL encourages explicit modularity. The language provides primitives which specify the interconnection of modules and their constituent sections.

Sections within a module follow a hierarchical organization, and allow the "imbedding" of sections within one another. During compilation, imbedding allows the recreation of the compile-time environment of previous sections. For example:

```
SECTION x;
SECTION x1 OF x;
SECTION x2 OF x1;
SECTION x3 OF x1;
```



```

      x
      |
      x1
     /  \
    x2   x3
  
```

Sections x2 and x3 have access to all global data defined in sections x1 and x. Section x1 may only reference data defined in section x. The name of the illustrated module is "x".

Modules communicate with one another through INTERFACE sections. Such sections specify the data that is exportable from a module. All other data is hidden. The keyword USES in the first statement of a module indicates the import of interface data from other modules. For example:

```
INTERFACE a USES b;
INTERFACE b;
SECTION c USES a,b;
```

Module b does not import any external data. The data in modules a and b may be exported since the keyword INTERFACE is used. At compilation time, module a imports the interface data from module b. The data in module c may not be referenced externally.

The explicit INTERFACE mechanism supplies excellent documentation. It also enhances program readability and maintainability.

### 3. The PROTEL Environment

Several tools have been created to assist the programmer in the development of DMS software. These include a PROTEL/DMS cross-compiler, a PROTEL/370 compiler, a linker, a loader, an emulator, a library system, and a debug system. The use of these tools will now be described, with particular emphasis on the compilers.

**3.1 Software Development Tools.** All DMS software resides on an IBM/370 under VM/370 in a common library which is managed by a sophisticated software control system. The PROTEL Library System (PLS) is a powerful tool which provides programmers with an environment for developing and maintaining large programs. Commands exist in PLS to allow the specification of the module and section interconnection structure. Other commands relate to the creation, modification,

compilation, and linking of these programs. PLS also supervises the interaction of several programmers working on a large project.

The PROTEL compilers execute on the /370, process source, and produce object and listing files. The object files contain object code and symbol table data used when imbedding.

DMS object files are combined into load files by the DMS linker in order to resolve external references and create modules. These load files can then be transported directly to the DMS machine via data link or stored on tape. The application then can be built in layers and loaded on the switch. Debugging and testing may also be performed on the /370 through the use of a DMS machine emulator.

On the DMS processor, the loader builds a program definition from constituent modules. This definition is then used to create processes which execute under control of a general purpose system designed specifically for DMS-100.

It is noteworthy that with the exception of the DMS emulator, all tools described above are implemented in PROTEL.

**3.2 Compiler Structure.** Both compilers are two-pass in-core language processors implemented using state-of-the-art compiler construction techniques. They are designed according to the modularity precepts of PROTEL. The first pass is common to both compilers and contains a LALR(1) parser [8]. This pass scans source statements and imbeds symbol table information from the object files of previously compiled sections.

The second pass of each compiler generates code from an intermediate parse tree representation of the input source using a recursive descent tree traversal algorithm. The compilers perform similar semantic checks.

Code generation is efficient in both compilers. The block structure of PROTEL maps well onto the DMS stack-oriented object code. In spite of the differences in CPU architectures, the machine code generated by the PROTEL/370 compiler is also quite compact. Both compilers contain peephole optimizers that improve code sequences within a small window of instructions [9,10].

### 4. Conclusions

PROTEL is a high-level language that is used as a development tool for the DMS-100 series of telephone switches. It was created to encourage modular system design and to enforce structured programming in keeping with state-of-the-art techniques. It provides high-level control constructs, powerful data structures, and strong type checking.

Since its inception in 1975, PROTEL has proven to be a highly successful language. To date, most development tools and all DMS-100 applications

software have been written in PROTEL. The successful introduction of the DMS-200 toll switch into commercial service [2] attests to the viability of the language and design approach.

PROTEL is also evolving as a general purpose system implementation vehicle. It has been used successfully at BNR in the implementation of a data base management system and a report generator for the IBM VM/370 system. Other applications are currently being planned.

#### Acknowledgements

We gratefully acknowledge the contributions of the following people in the development of PROTEL: J.C. Anderson, B. Beach, L. Fraser, N. Gammage, D. Lasker, and D. Sawyer. J. Kittle improved this paper through his careful criticism of the preliminary version.

#### References

- (1) Cashin, P.M., Gammage, N., Lasker, D., "DMS-200 Software Organization", International Conf. on Comm. Record 78, Toronto, June 4-7, 1978, section 32.4.
- (2) Munter, E., Patel, R., "DMS-200: A Toll Switch for the Digital World", Telesis, Vol. 6, No. 1, Feb. 1979, pp. 2-11.
- (3) Wirth, N., "The Programming Language PASCAL", ACTA Informatica, 1, 1971, pp. 35-63.
- (4) Habermann, A.N., "Critical Comments on the Programming Language PASCAL", ACTA Informatica, 3, 1973, pp. 47-57.
- (5) Lecarme, O., Desjardins, P., "More Comments on the Programming Language PASCAL", ACTA Informatica, 4, 1975, pp. 231-243.
- (6) Conradi, R., Holager, P., MARY Textbook, University of Trondheim, Trondheim, Norway, 1974.
- (7) van Winjngaarden, A., Mailloux, B.J., Peck, J.E.L., Koster, C.H.A., "Report on the Algorithmic Language ALGOL 68", Numerische Mathematik, 14, 1969, pp. 79-218.
- (8) Joliat, M.L., "Practical Minimization of LR(k) Parser Tables", Proc. IFIP Congress 74, North Holland Publishing Co., Amsterdam, 1974, pp. 376-380.
- (9) McKeeman, W., "Peephole Optimization", CACM, 8, July 1965, pp. 443-444.
- (10) Wulf, W., et al., The Design of an Optimizing Compiler, American Elsevier Publishing Co., N.Y., 1975.

<sup>1</sup> Trademark, Northern Telecom Limited.  
DMS (Digital Multiplex System) and DMS-100 are used interchangeably throughout this paper.