APL as a software design specification language

W. T. Jones* and S. A. Kirk†

Applied Mathematics and Computer Science Department, Speed Scientific School of Engineering, University of Louisville, Louisville, Kentucky 40208, USA

The purpose of this paper is to present a proposed extension of APL as a software design tool. The approach of system specification using a programming language is compared to a non-programming language PROPLAN which has been presented by Pengilly (1975). Illustrative examples given by Pengilly are translated into the proposed APL extension.

(Received June 1978)

A proposed extension of APL as a software design tool is presented. The approach of system specification using a programming language is compared to the non-programming language system PROPLAN presented by Pengilly (1975). Illustrative examples given by Pengilly are translated into the extended APL format.

Advocating the adoption of a programming language in software design is not intended to exclude the use of natural language or other formalisms and documentation aids such as HIPO which are commonplace in a software engineering environment. However, the importance and even necessity of specification languages which, when used in a design, can enhance the process of verifying correctness is increasing. This concern is further supported by the reliability that 'no programming philosophy will improve software reliability if the underlying system specifications are erroneous or have been incorrectly translated' (Belford and Taylor, 1976). Thus the selection or development of a software design language is inextricably intertwined with the specification verification problem. Falkoff (1976) has suggested the following criteria for the choice of a formal design language:

1. It should be easy to learn its basic use.
2. Formulations in the language should be suggestive and thought provoking.
3. It should allow meaningful formal manipulation of both expressions and larger program segments.
4. It should lead to the development of aesthetic criteria for judging the quality of a design.
5. It should be convenient for documentation.
6. It should be uncommitted to any particular technology or problem area.
7. It should be applicable in a consistent way to systems of any foreseeable complexity.
8. It should be executable on a machine.

To this list we would add

9. Lends itself to some form of specification verification technique.
10. Desirable, although not mandatory, that the design language be useful in other contexts such as ordinary programming applications to motivate learning.

The definition of a design specification language from the standpoint of the information systems analyst is 'a functional (non-procedural) programming language whose notation allows for quick, elegant and concise definition of the procedures used to input, store, process and output information'. The importance of a clean, rigorous specification to systems analysis is also becoming increasingly clear (Belford and Taylor, 1976; Liskov and Zilles, 1975). The *clarity* of software design must be well in hand before it can be assessed for correctness, efficiency, cost and other design constraints external to the system itself. A uniform, abstract notation is sorely needed for communication of the essentials of their design to other analysts *independent* of any particular implementation of the design.

In view of the above, APL or an extension thereof as a specification language has a number of attractive features. First, APL is in widespread usage and has been implemented to varying extents on many machines. APL is also mathematically amenable, highly symbolic and expressions are easy to write without temporary storage variables as well as without bothersome loops that index on array variables. Its powerful array operators lend themselves to functional (as opposed to procedural) statements. These features provide a recognised powerful design facility which can be used in the problem formulation and early design phase of a software project. For example, the use of APL in the design stage, augmenting FORTRAN, eliminated errors in the logical formulation stage (Kolsky, 1969). The primary features contributing to its success were the mathematical consistency and the inherent explicitness of APL program statements from a mathematical point of view.

All of the above listed criteria are met by APL. Objections to ease of learning with respect to software design applications should also take into account the fact that APL is much more than a design tool. The skills developed in the use of this facility can be transferred to non-design applications. Criterion 8 relates to the need for machine testing of designs. A formal language capable of expressing the design of a system at various levels of detail can be used directly for simulating the system at any stage in the top-down design process. Appropriate tables can be substituted for functions not yet detailed. Thus the opportunity is provided for 'live' catalogues on interactive systems in which functional units can be displayed and executed for gaining understanding of their behaviour (Falkoff, 1976). Criterion 9 seems to be particularly important since a formal definition of the APL operators has been used to provide a base for a deductive system for informally proving assertions about APL programs (Gerhart, 1972).

It may also, of course, be argued that APL has features which cause it to be undesirable as a software design language. An often mentioned fault is that of users taking advantage of the power of the language in the form of very complex statements or even perhaps complex single statement programs.These objections can be overcome by an extra measure of programming discipline which is required in the use of any high

*Now at Department of Computer and Information Sciences, University of Alabama in Birmingham, University Station, Birmingham, Alabama 35294, USA
†Present address: Tymshare Corporation, Chicago, Illinois

level language. In addition, however, the language, in its usual form, lacks the control structures that are known to enhance program design. The proposed extension of APL includes these needed features.

Proposed extension of APL

The following added features are proposed for APL to enhance its attractiveness as a design specification language:

1. Use of '_' (underline stroke) as a break character so the '∆' (delta) need not be relied upon as the *only* break character in APL would improve the readability of APL. The '∆' is much better used in APL variable names of increment (or decrement) variables than as a break character. (The sense of 'increment' here is that of the calculus in mathematics and not that of loop control in programming).
2. Higher order arrays are also needed. Arrays as they are defined in present APL implementations are said to be of the 1st order (or *level*). ∅-level arrays are regarded as scalars. A 1st-order array has as its element, ∅-order arrays (i.e. scalars). Likewise, a 2nd-order array may have as its elements 1st-order (i.e. 'ordinary') arrays. A straightforward, recursive definition of the ith-level array is simply one whose elements consist of $(i - k)$th-level arrays where k is an integer and $i \geqslant k \geqslant 1$. Thus a 5-level array may consist of 4-, 3-, 2-, or 1-level arrays and scalars. For the purposes of *notation* in APL, this distinction can be slight. However, should a language implementor decide to undertake this distinction in the extensions of APL proposed herein, this distinction would further complicate the *dope* vectors of the arrays. A revision in the dope vector to allow for a bit which tells whether or not *all* the elements of the vector have scalar elements is therefore desirable. Additional processing in the APL interpreter would be required to handle this extra bit.
3. Concerning the general usage of the dimensional indicators mode of the APL operators, *multi*-dimensional indicators are useful where single-dimensional indicators are presently used as in *scan* and *reduce*. (In this case the result of the operation is an array *of* arrays (a higher level array) as opposed to an array *simpliciter* (a 1-level array). The proposed extension also includes dimensional indicators for the *grade-up* and *grade-down* operators to indicate which dimensions should be used to do the 'sorting'.
4. A further extension of the grade-up and grade-down operators can be implemented by allowing a Boolean vector to be written to the left of the operator to denote the *reverse* of the grade operator designated, i.e. a *zero* would indicate that the grade-up is to be taken when the grade-down is specified and that the grade-down is to be taken when the grade-up is specified. A *one* would have a null effect and merely serve as padding to allow for the determination of which dimension(s) each bit in the Boolean vector applies to via the position of the bit. This Boolean vector addition to the grade operators is essential in the case of ordering or sorting with the grade operators when used with the multi-dimensional specifier proposed above. The reader is referred to the examples of this notation given in the last section.
5. The use of the 'A' symbol (lamp) to permit end-of-line comments with statements is recommended. This may prove difficult to implement given that APL interprets statements and lines of statements from right to left.
6. A default to the *last* dimension for *all* operators which are capable of taking dimensional indicators when they are written with*out* dimensional indicators is also desirable.
7. A follow-up to the proposal to introduce higher order arrays, assignment (symbols '←') should be allowed to operate on an array of arrays.
8. The following structured programming control flow concepts may easily be represented in a symbolic fashion by allowing for the *dy*adic usage of the goto operator (symbol '→') (Kelley, 1972).

   IF. . . THEN. . . construction may be represented by . . . → . . . in APL.

   IF . . . THEN . . . ELSE . . . construction may be represented by . . . → . . . ↛ . . . in APL.

   Likewise:

   . .˜. ↛ . . . may be used to express
   IF. . . ELSE. . . which is usually written as
   IF NOT(. . .) THEN. . . .

   There are further possibilities which would conceivably require a recursion control stack to implement such as

   . . . → . . . → . . . → . . .
   and . . . ↛ . . . ↛ . . . ↛ . . .
   and . . . → . . . → . . . ↛ . . . .

   Other proposals for the inclusion of these control structures have been made by Lim and Lewis (1975).

In the following section the utility of this proposed extended version of APL is demonstrated by examples which are translations of examples given by Pengilly (1975) using the Programming Planning language for software design.

Proposed revisions in Pengilly's notation

1. There should be a sharp distinction drawn in Pengilly's systems attributes; to wit: the *parameters* of the system v. the *constants* of the systems. In particular,
   a, b, c, d, e, f, g, h, and i are the system parameters and q and l are the system constants in his example system.
2. Different input modes for system parameters and constants are suggested:
   file input (symbol ←) for system parameters and
   operator input (symbols ←) for system constants
   where the above symbols are written to the *right* of their respective operands.
   It should be noted that the above two symbols are to be used in place of Pengilly's ρ (rho) which is a symbol having a completely different meaning in APL.
   For output the following is used:
   ← for printing (instead of π (pi))
   ← for filing (instead of ∅)
   The above symbols are written to the *left* of the respective operands (as distinguished from their counterpart *in*put operators.) Again 'pi' means something utterly different in APL. (The Greek letter π is not available in the APL character set.)
3. Eliminate the slash (symbol '/') notation as Pengilly uses it. Uniqueness of a systems parameter within another should be specified without so noting at each occurrence of a variable which has the unique parameter.
4. Elimination of the at-sign (symbol '@') as a dummy symbol.
5. The following comparisons should be kept in mind in adapting Pengilly's approach to design in APL:

ATTRIBUTE:	dimension of an array ::
ELEMENT:	element of an array

   Also,

ATTRIBUTE:	aspect of a datum ::
ELEMENT:	a datum.
6. From the systems viewpoint, the following analogy to the two key terms of Pengilly's is made:

ATTRIBUTE:	system parameter ::
ELEMENT:	system variable.

7. For the cataloguing procedure (#7) the following variable needs to be defined:

$C[,,,d/e/f,/,]$

using Pengilly's notation.

Revised procedures

```
1.     ∇ BASIC_INPUT
  [1] (R,Q,K)  ⍇ A R (∅); data file input
  [2] (Q,L) ←  ⎕  A R(∅) cont.; system constant input
                  by the operator
       ∇

2.     ∇ TAKE_STOCK
  [1] (K−H) → (⍇ST [⍋[1]ST])          A P(1)
  [2] S⍇                               A R(2)
  [3] Z ← I − S                        A P(2)
  [4] Y ← (+/[1,2]Z) × J × E           A P(2) cont.
  [5] ⍇Z⍋[5,2,1]Z]                     A O(2)

3.     ∇ DAILY_STOCK_UPDATE
  [1] (A,L,O)⍇                                      A R(3)
  [2] ⍇ +/[1,2,3,4]A                                A R(3) cont.
  [3] ⍇ +/[1,2]L                                    A R(3)  ”
  [4] ⍇O                                            A R(3)  ”
  [5] I←I,[6](I + ( +/[6,7]) − (+/[3]L))            A P(3)
  [6] Y ← +/[1,2]L × J                              A P(3) cont.
  [7] ⍇Y                                            A P(3)  ”
  [8] ⎕ ← Y                                         A O(3)
       ∇

4.     ∇ ISSUE_PRICE
  [1] (B,BP)⍇                                          AR(4)
  [2] (B∧(((((I−B)×J+(B×BP))÷I)−J)÷J)−Q) →
                    (J←((I−B)×J+B×BP)÷I)              A P(4)
  [3] ⎕ ← J                                            A O(4)
       ∇

5.     ∇ ORDER
  [1] SC⍇                                              A R(5)
  [2] (~(+/[1,2]I−R+U)) → (OT←Q) ↛
                              (OT ← SC)                A P(5)
  [3] ⎕ ← OT[∆[1]OT]                                   A O(5)
       ∇

6.     ∇ PAY_SUPPLIER
  [1] ⍇O⍇                                           A R(6)
  [2] ⍇OP⍇                                          A R(6) cont.
  [3] (A∧O∧B)∧((BP−(1,1,1,¯1) ⇑ OP)÷
      (BP−L)) → (⎕←X[⍋(2,3,1)X];⍇X[X←B×BP)
      ↛ (⎕ ← (A,O,B,BP[⍋[2,3,1]BP])                 A P(6)
       ∇

7.     ∇ VALUE_STORES
  [1] V ← (+/[1,2]I)×J                        A P(7)
  [2] ⎕ ← V[⍋[1,2,3]V]                        A O(7)
  [3] ⎕ ← +/[1,2] I[⍋[3,4,5]I]                A O(7) cont.
  [4] ⎕ ← J                                   A O(7) cont.
  [5] ⎕ ← +/[1,2,3]V                          A O(7) cont.
       ∇

8.     ∇ CATALOGUE
  [1] ⎕ ← C[⍋(1,2,3]C]                        A O(8)
       ∇
```

Relationship to data base system design

The problem of data base design is related to the problems addressed by Pengilly and the present authors. However, the problem context differs wherein these two approaches are seen as appropriate. In the case of data base design, the problem is characterised by the need for a conceptual schema for a large centralised collection of inter-related files which are accessed by a number of application programs for different purposes. It is also usually an objective that the conceptual schema together with the data manipulation language be designed such that new unforeseen application programs can be quickly developed requiring the generation of new data relationships from some specified subschema.

On the other hand, while the software design approach of Pengilly and this paper can clearly assist in the above type of problem context, it would appear more directly appropriate for the functional specification of software systems in a more general way than the more specific and narrow case described above requiring the data base management system.

Conclusions

A modified version of APL has been shown by illustrative examples to be potentially useful as a software specification tool. The notable advantages of APL in this regard are its functional orientation for specifying design functions and its mathematical amenability for proof of correctness techniques. The most obvious disadvantage, of course, is the fact that the sheer power of the language, which is a strong advantage from the standpoint of top-down functional design, is much more susceptible to abuse. If APL is to realise its potential in the software specification domain a more stringent enforcement of appropriate programming standards at the software design establishment must be implemented.

It is also noted that APL may be most useful at the very earliest stages of design and later augmented with other design tools.

References

BELFORD, P. C. and TAYLOR, D. S. (1976). Specification Verification—A Key to Improving Software Reliability, *Proceedings of Symposium on Engineering*, Vol. XXIV, pp. 83-96.

FALKOFF, A. D. (1976). Criteria for a System Design Language, in *Software Engineering: Concepts and Techniques*, Peter Naur *et al*, (eds.), Petrocelli/Charter, New York, pp. 226-231.

GERHART, S. K. (1972). *Verification of APL Programs*, Carnegie Mellon University, AD754856, November 1972.

KELLEY, R. A. (1972). Structured Programming Language for APL, *IBM Technical Disclosure Bulletin*, Vol. 15, pp. 1397-1398.

KOLSKY, H. G. (1969). Problem Formulation using APL, *IBM Systems Journal*, Vol. 8 No. 3, pp. 204-217.

LIM, A. L. and LEWIS, G. R. (1975). Structured Programs in APL, *The Computer Journal*, Vol. 18, pp. 140-142.

LISKOV, B. H. and ZILLES, S. N. (1975). Specification Techniques for Data Abstractions, *IEEE Transactions on Software Engineering*, Vol. 1, pp. 7-19.

PENGILLY, P. J. (1975). An Approach to Systems Design, *The Computer Journal*, Vol. 18, pp. 8-12.