[22] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson, "A provably secure operating system: The system, its application, and proofs," SRI Project 4332, Final Rep., Feb. 1977.

[23] E. I. Organick, *The MULTICS System: An Examination of Is Structure*. Cambridge, MA: MIT Press, 1972.

[24] D. L. Murphy, "Storage organization and management in TENEX," in *Proc. AFIPS 1972 Fall Joint Comput. Conf.*, vol. 41. Montvale, NJ: AFIPS Press, pp. 23-32.

[25] J. H. Saltzer, "Protection and the control of information sharing in MULTICS," *Commun. ACM*, vol. 17, pp. 388-402, July 1974.

[26] M. D. Schroeder, "Cooperation of mutually suspicious subsystems in a computer utility," Ph.D. dissertation, MIT, Sept. 1972, also available as Project MAC TR-104.

[27] D. L. Chaum and R. S. Fabry, "Implementing capability-based protection using encryption," Univ. of California, Berkeley, Memorandum ERL M78/46/UCB, July 1978.

[28] E. I. Organick, *Computer System Organization: The B5700/B6700 Series*. New York: Academic, 1973.

[29] B. W. Lampson, "Dynamic protection structures," in *Proc. AFIPS 1969 Fall Joint Comput. Conf.*, vol. 35. Montvale, NJ: AFIPS Press, pp. 27-38.

[30] B. W. Lampson, "Protection," in *Proc. 5th Ann. Princeton Conf.*, Princeton Univ., Mar. 1971, pp. 437-433.

[31] D. D. Chamberlain, J. N. Gray, and I. L. Traiger, "Views, authorization, and locking in a relational data base system," in *Proc. AFIPS 1975 Nat. Comput. Conf.*, vol. 44. Montvale, NJ: AFIPS Press, pp. 425-430.

[32] P. P. Griffiths and B. W. Wade, "An authorization mechanism for a relational database system," *ACM Trans. Database Syst.*, vol. 1, pp. 242-255, Sept. 1976.

[33] D. H. Vanderbilt, "Controlled information sharing in a computer utility," Ph.D. dissertation, MIT, Oct. 1969, also available as Project MAC, TR-67.

[34] R. S. Fabry, "Capability-based addressing," *Commun. ACM*, vol. 17, pp. 638-642, Aug. 1974.

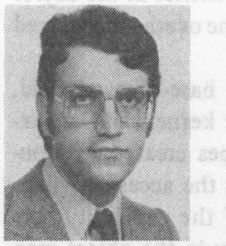
[35] H. W. Lawson and L. Blomberg, "The Data Saab FCPU Microprogramming language," *SIGPLAN Notices*, vol. 9, pp. 87-96, Aug. 1974.

[36] H. W. Lawson and R. Malm, "A flexible asynchronous microprocessor," *BIT*, vol. 2, pp. 165-176, June 1973.

[37] W. A. Wulf, R. L. London, and M. Shaw, "An introduction to the construction and verification of Alphard programs," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 253-265, 1976.

[38] P. A. Janson, "Using type extension to organize virtual memory mechanisms," Ph.D. dissertation, MIT, Sept. 1976.

[39] B. W. Lampson, R. Needham, B. Randall, and M. Schroeder, "Protection, security, reliability," *SIGOPS Oper. Syst. Rev.*, vol. 11, pp. 12-14, Jan. 1977.



Virgil D. Gligor received the B.Sc., M.Sc., and Ph.D. degrees in electrical engineering and computer science in 1972, 1973, and 1976, respectively, all from the University of California, Berkeley.

In 1976, he joined the Department of Computer Science at the University of Maryland as an Assistant Professor. Since 1977 he has also been associated with the Computer Systems Group of Burroughs Corporation. He has lectured in computer science at the University of California at Santa Barbara (1974-1976) and at the Institutul Politechnic Bucuresti in Romania (1972). His research interests include reliability and integrity of operating systems, computer architecture, and distributed data base systems.

Program Derivation Using Data Types: A Case Study

CARLOS J. P. LUCENA, MEMBER, IEEE, AND TARCISIO H. C. PEQUENO

Abstract—The present paper discusses some issues in program synthesis by relating the idea of systematic program derivation with the concepts of data type and correctness of data representation. The notion of an incomplete definition of a data type at a high level of abstraction is introduced. The ideas are illustrated through an example previously discussed in the literature by D. Gries.

Index Terms—Correctness of data representation, data types, program derivation, program schema, program specification, program synthesis.

I. Introduction

THE state of the art in the area of program verification, begun in the late 1960's by Floyd [1], is now reaching a stage in which many of the existing results on analysis of programs are starting to be transferred to practice through research on software engineering. Efforts have now turned to the goal of providing a methodology for the systematic synthesis of programs. In fact, many people are presently working on the problem of deriving a program from a given program specification. Our goal in this paper is to contribute to the better understanding of this process.

Depending on the system of notation which is used to express the program specification, the above problem can be seen as deriving a practical program either from an inefficient one (*vis-à-vis* the current computer architectures) or from a less operational program statement. It is possible to include in the first category the works by Burstall and Darlington [2], Bauer [3], and Arsac [4]. These authors have chosen to specify programs through recursion equations. The recursive form being well adapted to manipulations (transformations) allows for the establishment of a set of rules to transform programs (specifications), written for maximal clarity, into practical or ade-

Manuscript received November 1, 1978; revised May 18, 1979.
The authors are with Departamento de Informatica, Pontificia Universidade Catolica, Rio de Janeiro, Brazil.

quately efficient programs. Most of these efforts are being accompanied by the development of software systems for program development.

The basic approach taken for the synthesis of the usual Algol-style form of a program requires the use of nonprocedural program specifications. One approach to the problem solution can be in this case the use of Dijkstra's idea on constructive programming together with Hoare's rules for the verification of Algol-like programs. Dijkstra explores the idea of predicate transformers [5], [6] and proposes a methodology for the derivation of programs from their post-conditions and pre-conditions (specifications). Manna, together with several collaborators, e.g., [7], developed work on a spectrum ranging from the automatic synthesis of simple programs specified by their input and output assertions to the design of an interactive system in which the computer takes the more straightforward steps on its own, while the human guides the machine in the more creative ones.

Since programmers will not be inventing completely new programs all the time, efforts are being made to provide program-writing systems with the capability to learn from old programs. Gerhart [8], [9] has been working on the compilation of a handbook of program schemas that can be abstracted from most current programming applications and that can be used for the synthesis of more complex programs.

In the above, we have briefly summarized some of the current ideas on program synthesis. While these efforts are still taking place, an overwhelming majority of programmers continue to write programs in Algol-like languages with very little understanding about the objects they are producing. It is important to explain to these programmers some of the more established ideas on the program synthesis process.

Gries [10] contributed to this purpose by illustrating some of Dijkstra's ideas while applying them to a reasonably typical programming example. However, our attention was brought to the fact that the current practice of separating the algorithm and the data aspects of a program in the program development process (program = algorithm + data structures [11]) was not taken into consideration in Gries example. In fact, the synthesis counterpart of Hoare's ideas on the correctness of data representations [12] can be found, in a formalized way, only in conjunction with Burstall's work on recursive programs [2]. Works by Liskov and Zilles [13], Dennis [14] and Wulf [15], have dealt with some of the advantages (mostly from the point of view of programming practice) of program development through the use of data abstractions. These works are very important for the discussion of the synthesis of Algol-like programs.

In what follows we discuss an alternative solution to the example proposed by Gries in [10]. In explaining the synthesis ideas we use the language of first-order predicate calculus which is adequate for a critical approach and discussion of the problem. We leave out all considerations of efficiency in order to concentrate on the viability of integrating the notion of a data type specification with Hoare's or Dijkstra's style of proof rules. Our intent is not to propose a methodology for program derivation but rather to try to contribute to the understanding of the program derivation process.

II. The Line-Justifier Example: Synthesis of the Program Schema

As stated above, we will present our view on the program synthesis process as we discuss the derivation of a program that uses the concept of an abstract data type, as suggested for example by Gries. The emphasis will be on the interaction between the derivation of a program schema and the construction of a data representation model for the program.

The suggested derivation procedure can be summarized in the following steps.

Step 1: Choice of a suitable data type; the operations of the data type provide a language in order to talk about the problem domain.

Step 2: Statement of some properties of the type (those that seem appropriate to derive a program schema).

Step 3: Derivation of the program schema.

Step 4: Choice of a data representation and definition of the data type in terms of this representation.

Step 5: Proof of the correctness of the representation (consistency with the properties of the data type).

Step 6: Derivation of the "cluster" [13] (programming mechanism that implements the representation).

A. The Problem Statement

A line-justifier is the part of a text editor that inserts blanks between words in a line in a way that avoids the existence of blanks after the last word or before the first word in the line. We want to construct a line-justifier program according to the following specifications:

1) It accepts as input a numbered left-justified line having more than one word in which there will be just one blank between words and possibly several blanks after the last word.

2) It will produce as output a justified line, that is, a line in which the extra blanks to the right of the last word will have been distributed in the spaces between the words on the line. The difference between the number of blanks in two arbitrary intervals will be at most one. When there is a difference, the number of blanks between words will be the same up to a given word in the line; and after this word the number of blanks between words will again be uniform, but there will be either one more or one less than the previous number of blanks. For aesthetic reasons the even lines will have more blanks at the beginning of the line and the odd lines more blanks towards the ends.

B. The Type Line

As indicated in the problem statement, the line-justifier program will manipulate objects called lines. For that reason we shall define a data type *line* formed by the set of these abstract objects (ambiguously also called lines).

The informal problem statement refers to restriction associated with lines that are to be accepted by the program. It also refers to the characteristics of the output lines and distinguishes between the treatment to be given to odd and even lines. These facts suggests the definition of the following functions and predicates:

1) A unary predicate $\underline{\text{is-initial}}(x)$ determines if a line x satisfies the input restrictions.

2) A binary predicate $\underline{\text{is-just}}(x,y)$ determines if a line y is the result of the justifications of a line x.

3) A pair of unary operations for line justification: The first will perform the justification by inserting a larger number of extra blanks to the left of the line and the second by inserting a larger number of extra blanks to the right. They will be called $\underline{\text{just-left}}(x)$ and $\underline{\text{just-right}}(x)$, respectively.

4) A unary predicate $\underline{\text{even}}(x)$ determines if the number associated with a line x is even.

Of course, the choice of the above functions and predicates will affect the initial form of the program. Our intention in making these choices has been to adhere naturally to the informal problem statement. As stated in the introduction, no consideration is given in the example to problems of efficiency. Nevertheless, it should be noted that a choice made at this point is, by no means, final. If one aims at efficiency, one can always apply the adequate transformations and change the initial version of the program into a more efficient one. These transformations could, for instance, combine the effects of just-left and just-right in a single operation [3], [4].

The above set of operations defines a first-order language L = <is-initial, is-just, even, just-left, just-right>, which we will use to talk about lines. The informal program specification given in Section II-A requires that the above described operations satisfy the following self-explanatory axioms.

Axiom 1:

$$\underline{\text{is-initial}}(x) \wedge \underline{\text{even}}(x) \rightarrow \underline{\text{is-just}}(x, \underline{\text{just-left}}(x)).$$

Axiom 2:

$$\underline{\text{is-initial}}(x) \wedge \neg\, \underline{\text{even}}(x) \rightarrow \underline{\text{is-just}}(x, \underline{\text{just-right}}(x)).$$

It is interesting to note that the above axioms define a class of data types. In fact, to be able to define the type completely, we would have to state some further properties about the operations and predicates which would capture the details of the justification method contained in the informal problem definition. We will see that we will be able to do that when we associate a particular representation to the type (an interpretation of the specification). The properties expressed through the given axioms are sufficient for the derivation of a program schema which will, as a first approximation, solve a class of line-justification problems.

C. *Program Schema Derivation*

Using the L-language defined above, the program specification can now be restated in the following manner:

$$\{\underline{\text{is-initial}}(x)\}\ P(x,y)\ \{\underline{\text{is-just}}(x,y)\}$$

In the input assertion (pre-condition) $\{\underline{\text{is-initial}}(x)\}$, variable x is an input variable. In the output assertion (post-condition) $\{\underline{\text{is-just}}(x,y)\}$ the variable y is both a program variable and an output variable. Our goal, at this point, is to derive the program $P(x,y)$.

For an x, such that, $\underline{\text{is-initial}}(x)$ is true, the two axioms given above can be restated as

$$[\underline{\text{even}}(x) \rightarrow \underline{\text{is-just}}(x, \underline{\text{just-left}}(x))]$$

$$[\neg\, \underline{\text{even}}(x) \rightarrow \underline{\text{is-just}}(x, \underline{\text{just-right}}(x))]$$

The axioms so expressed suggest the use of the ifthenelse, with $\underline{\text{even}}(x)$ as the predicate. Thus, we have the program of the following form:

```
            {is-initial(x)}
P(x,y) ≡ if even(x)
            then S_1(x,y)
            else S_2(x,y)
         fi
            {is-just(x,y)}
```

The ifthenelse verification rule reads as follows [16]:

$$\frac{\{Q \wedge t\}\ S_1\ \{R\},\ \{Q \wedge \neg t\}\ S_2\ \{R\}}{\{Q\}\ \underline{\text{if}}\ t\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}}\ \{R\}}$$

In this particular case, we have

Q = is-initial(x);
t = even(x);
R = is-just(x,y).

In program analysis, verification rules are applied by checking the pre- and post-conditions of the antecedent of the rule to allow the statement of the expression used as its consequent. When deriving a program we must invert this process. The application of the rule consists now of using the expression proposed as the consequent to derive the pre- and post-conditions of the program segments structured by the control mechanism defined by the rule. Using the above rule for synthesis purposes, we can state:

i) $\{\underline{\text{is-initial}}(x) \wedge \underline{\text{even}}(x)\}\ S_1(x,y)\ \{\underline{\text{is-just}}(x,y)\}$

and

ii) $\{\underline{\text{is-initial}}(x) \wedge \neg\,\text{even}(x)\}\ S_2(x,y)\ \{\underline{\text{is-just}}(x,y)\}$.

By *modus ponens* of Axiom 1 with the pre-condition of i) above, we have

$$\{\underline{\text{is-just}}(x, \underline{\text{just-left}}(x))\}.$$

Analogously, for ii) we can write

$$\{\underline{\text{is-just}}(x, \underline{\text{just-right}}(x))\}.$$

We are now ready to apply the assignment axiom [16], which reads

$$\{Q(x, f(x))\}\ y := f(x)\ \{Q(x,y)\}.$$

Its application will produce

$$S_1(x,y) \equiv y := \underline{\text{just-left}}(x)$$

and

$$S_2(x,y) \equiv y := \underline{\text{just-right}}(x)$$

which implies the following program:

```
            {is-initial(x)}
P(x,y) ≡ if even(x)
            then y := just-left(x)
            else y := just-right(x)
         fi
            {is-just(x,y)}
```

This program schema can be encoded in the following CLU-like [13] notation:

```
line-justifier = procedure (x:line) returns (line)
   y:line;
      ¢ is-initial(x) ¢
   if line$even(x)
         then y:= line$just-left(x)
         else y:= line$just-right(x)
   fi
   return (y);
   ¢ is-just(x,y) ¢
   end line-justifier
```

The reason for encoding the program schema in CLU is that we shall later make use of CLU's cluster mechanism for expressing data types. We must, however, call the reader's attention to the fact that we are not bound to any particular programming language. A programmer, in the context of our work, can choose to use any control or data structure, providing he is able to state its axioms formally.

III. The Line-Justifier Example: Definition of the Data Representation

So far we have abstracted some properties of any representation of the object line. We are now going to associate a specific model (representation + operations) with the theory defined by the axioms in language L.

By the problem definition, our program receives as input a line expressed in a given representation and produces as output a line expressed in the same representation. Therefore the line representation is an integral part of the problem definition. We are going to solve the proposed problem through the use of two different representations. The first one exactly matches the specific problem; the second is similar to that adopted in Gries' solution [10].

We are initially going to think of a line as a six-tuple of natural numbers, having the following components:

p	number of blanks in the leftmost intervals of the line.
q	number of blanks in the rightmost intervals of the line.
t	index of the word after which the number of blanks changes.
n	number of words on a given line.
s	number of extra blanks at the end of the line.
z	line number.

Note that the program being developed does not handle the text itself. We, as Gries did, suppose that the text was preprocessed to produce a representation that contains only the aspects directly related to the problem. In fact, Gries includes some extra information in his representation, and we will do approximately the same in our second choice of representation.

The domain of the type line, which we will also call line, will be the following subset of N^6:

$$\underline{\text{line}} = \{<p,q,t,n,s,z> \in N^6 \mid t \leqslant n \wedge n > 1 \wedge |p-q| \leqslant 1 \wedge p \geqslant 1 \wedge q \geqslant 1\}.$$

We now define the operations of line in our model. For that purpose we will use the variables

$$x = <p,q,t,n,s,z>$$

and

$$x' = <p',q',t',n',s',z'>$$

The proposed definitions are the following:

1) $\underline{\text{is-initial}}(x) \overset{\text{df}}{=} \quad p = q = 1 \wedge t = n;$

2) $\underline{\text{even}}(x) \overset{\text{df}}{=} z$ is even;

3) $\underline{\text{just-left}}(x) \overset{\text{df}}{=} <p + s \div (n-1) + 1,\; q + s \div (n-1),\; \text{mod}(s,(n-1)) + 1,\; n, 0, z>;$

4) $\underline{\text{just-right}}(x) \overset{\text{df}}{=} <p + s \div (n-1),\; q + s \div (n-1) + 1;\; n - \text{mod}(s,(n-1)),\; n, o, z>;$

5) $\underline{\text{is-just}}(x,x') \overset{\text{df}}{=} \quad s' = 0 \wedge s + p.(t-1) + q(n-t) = p'.(t'-1) + q'.(n'-t') \wedge n=n' \wedge z=z'$

The computation used for the definitions of the operations was borrowed from Gries [10].

A. *Verification of the Representation*

To verify the proposed data representation we need to prove first that its functions and relations are well defined in the specified domain. The relations are obviously well defined since they are expressed in terms of the operations and predicates defined over the naturals numbers. We must then start by proving the closure of the functions which alter objects of type line.

The next step is to prove that we have defined a model (representation) of the theory (type) proposed in Section II. In other words, we need to verify that the model satisfies axioms 1 and 2.

B. *Proof of the Closure Property*

Although just-right and just-left are applied only once in the present example, the closure of these operations will be checked to guarantee that the output predicate is applicable.

$\underline{\text{just-left}}(x) = x' = <p',q',t',n',s',z'>$

1) $p' = p + s \div (n-1) + 1$. Since $n > 1$, $n-1$ is a natural number different from 0, therefore $s \div (n-1)$ is a natural number. Since $p \in N$ and N is closed under addition, $p' \in N$ and $p' \geqslant 1$. □

2) We can analogously conclude that $q' \in N$ and since $q' = q + s \div (n-1)$ we have $p' = q' + 1$ or $p' - q' = 1$ and therefore $|p' - q'| \leqslant 1$. We also have $q' \geqslant 1$. □

3) $t' = \text{mod}(x,(n-1)) + 1$. Since $\text{mod}(s,(n-1)) < n-1$ we have that $t'-1 < n-1$ or $t' < n$. Since $n' = n$, then $t' < n'$. □

4) Since $n' = n$, $s' = 0$, $z' = z$ we trivially have that $n', s', z' \in N$ and $n' > 1$. □

The closure of just-right can be verified through the same procedure.

C. *Proof of the Satisfiability Property*

We will now prove that we do have a model that satisfies the given theory. We will show that the model satisfies Axiom 1 of Section II and will leave the proof of satisfiability of Axiom 2 to the reader.

Axiom 1 is

$$\underline{\text{is-initial}}(x) \wedge \underline{\text{even}}(x) \rightarrow \underline{\text{is-just}}(x, \underline{\text{just-left}}(x)).$$

Assume $\underline{\text{even}}(x) \wedge \underline{\text{is-initial}}(x)$, we can write

z is even, $p = 1, q = 1$ and $t = n$.

Now, let $x' = \underline{\text{just-left}}(x)$, that is,

$$\begin{aligned} p' &= p + s \div (n-1) + 1 = 2 + s \div (n-1) \\ q' &= q + s \div (n-1) = 1 + s \div (n-1) \\ t' &= \text{mod}(s, (n-1)) + 1 \\ n' &= n, s' = 0, z' = z. \end{aligned}$$

We must show that $\underline{\text{is-just}}(x, x')$ is true. Since $n' = n$ and $z' = z$, we need only verify that

$$s + p(t-1) + q(n-t) = p'(t'-1) + q'(n'-t').$$

Let us now replace in the right-hand side of the above equality the value of p', t', q', n':

$$\begin{aligned} &(2 + s \div (n-1))(\text{mod}(s,(n-1)) + 1 - 1) \\ &\quad + (1 + s \div (n-1))(n - \text{mod}(s,(n-1)) - 1) \\ &= 2\,\text{mod}(s,(n-1)) + s \div (n-1)\,\text{mod}(s,(n-1)) + (n-1) \\ &\quad + (n-1)\,s \div (n-1) - \text{mod}(s,(n-1)) \\ &\quad - s \div (n-1)\,\text{mod}(s,(n-1)) = \text{mod}(s,(n-1)) + (n-1) \\ &\quad + s \div (n-1)(n-1) = s - s \div (n-1)(n-1) + (n-1) \\ &\quad + s \div (n-1)(n-1) = s + n - 1. \end{aligned}$$

Replacing the values of p, q, and t in the left side of the equality above, we get

$$s + 1\,(n-1) + 1\,(n-n) = s + n - 1.$$

In Section II-B we formalized through Axioms 1 and 2 some aspects of the informal problem definition. These aspects were chosen to be those informally considered necessary for the derivation of a program schema that captures the general idea suggested by the informal problem definition. Therefore, Axioms 1 and 2 were not meant to define the type line completely. As we moved to the representation level our model was intentionally required to satisfy some more properties which refer to the additional program requirements contained in the problem definition.

Following our approach the complete type specification comprises both the so-called abstract and representation levels [17], [18] and each cannot be used independently to derive a program to solve the problem completely. Therefore, the notion of modularity is not used here in the usual way. This is because we relax the requirement of having a complete machine at some levels.

D. Derivation of the Cluster for the Data Representation

Having shown that we have a legitimate model, we need now to produce a programmed version of the model. For that purpose we are going to use the cluster mechanism, as proposed in [13]. The cluster will contain a representation (global to the procedures in the cluster) whose invariant is to be a line (defined above). The invariant has been verified in Section III-C and will remain valid if the operations in the cluster follow their respective definitions. For the various procedures implementing the operations within the cluster, the post-condition is the definition of the operation and the pre-condition is the invariant line.

The derivation of the programs implementing the various operations is a very simple exercise for the present example. All that needs to be done is the successive application of the rules of assignment and concatenation to the procedures' post-conditions. Since we have already derived in Section II a program segment by using the assignment axiom, we will omit the straightforward discussion.

The cluster program has, for the present case, the following form:

```
line = cluster is even, just-left, just-right;
rep = record (p:integer; q:integer; t:integer; n:integer;
              s:integer; z:integer);
create
  l:rep;
even = oper (x:cvt) returns (boolean);
         return(EVEN(z));
         end even;
just-left = oper (x:cvt) returns (cvt);
              l.q := x.a + x.s÷(x.n-1),
              l.p := l.q + 1;
              l.t := mod(x.s, (x.n-1)) + 1
              l.s := 0; l.n := x.n; l.z := x.z;
              return(l);
              end just-left;
just-right = oper (x:cvt) returns (cvt);
              l.p := x.p + x.s÷(x.n-1);
              l.q := l.p + 1;
              l.t := x.n-mod(x.s, (x.n-1));
              l.s := 0; l.n := x.n; l.x := x.z;
              return(l)
              end just-right;
        end cluster
```

We have then derived a program for the given specification, that is, a program that captures what was stated by the original problem definition.

IV. Change of Data Representation

The derivation of the programmed data representation, as it was proposed before, was extremely simple because we adopted a minimal (in some sense) configuration for the representation data space. It contained just the necessary elements for the satisfaction of the problem specification. In Gries' example [10] some extra features were added to the representations: an array of indices is used to indicate where each word in the line begins. The reader who wants to compare the two solutions must pay attention to the fact that the meaning of the natural numbers p and q in Gries' example is slightly different from ours, since in his example they stand for the number of blanks to be inserted between the words. It is interesting to show how our program can be modified so that we can use a similar model.

Let A be the set of all arrays of natural numbers and b a variable ranging over its domain. We need now to restate the domain line.

Let us now call line, the following set

$$\underline{\text{line}} = \{<p,q,t,n,s,z,b> \in N^6 \times A \mid t \leqslant n \wedge n > 1 \wedge |p-q| \leqslant 1 \wedge |b| = n \wedge b_1 = 1 \wedge (1 \leqslant i < t) \to b_{i+1} > b_i + p \wedge (t \leqslant i < n) \to b_{i+1} > b_i + q\},$$

where p,q,t,n,s,z have the same meaning as before and b is an array which stores the position of the beginning of each word in the line.

We shall now define the operations of the new model. The operations is-initial and even will have the same definitions as before. The operations just-left and just-right will now be defined as follows:

$$\underline{\text{just-left}}(<\bar{x},b>) = <\bar{x}',b'>$$

and

$$\underline{\text{just-right}}(<\bar{y},b>) = <\bar{y}',b'>$$

where $\bar{x}'$ and $\bar{y}'$ are to be computed as in the previous definitions of just-left and just-right and b', for both definitions, will have the following meaning:

$$b' = \begin{cases} \text{for } 1 \leqslant i \leqslant t', & b_i' = b_i + (p'-p)\cdot(i-1) \\ \text{for } t' < i \leqslant n, & b_i' = b_i + (p'-p)\cdot(t'-1) + (q'-q)\cdot(i-t') \end{cases}$$

Finally, given $x = <p,q,t,n,s,z,b>$ and $x' = <p',q',t',n',s',z',b'>$ we can define the predicate is-just

$$\underline{\text{is-just}}(x,x') \overset{\text{df}}{=} s' = 0 \wedge s+p(t-1)+q(n-t) = p'(t'-1) + q'(n'-t') \wedge n=n' \wedge z=z' \wedge (1 \leqslant i \leqslant t') \to b_i' = b_i + (p'-1)(i-1) \wedge (t' < i \leqslant n) \to b_i' = b_i + (p'-1)(t'-1) + (q'-1)(i-t')$$

The verification of this representation can be done in a way similar to that shown above. It is too tedious and will be omitted.

A. *Derivation of the Cluster for the New Representation*

We have to encode the new representation as a cluster of procedures. As mentioned before, the procedures which implement the operations can be derived from the definitions of the operations. We shall illustrate this procedure by deriving the operation just-left. This way we have the opportunity of using for the first time in this paper the proof rule for an iteration.

Input variables: $<p,q,t,n,s,z,b> = \bar{x}$.
Program variables: $<p',q',t',n',s',z',b'> = \bar{y}$.
Pre-condition: T.
Post-condition:
(a) $p' = p+s\div(n-1)+1 \wedge q' = q+s\div(n-1) \wedge t' = \mathrm{mod}(s,(n-1))+1, n'=n, s'=0, z'=z \wedge$
(b) For $1 \leqslant i \leqslant t', b_i' = b_i + (p'-1)(i-1) \wedge$
(c) For $t' < i \leqslant n, b_i' = b_i + (p'-1)(t'-1) + (q'-1) i-t')$.

We shall call Q the predicate that express the post-condition and will split it into three components, such that

$$Q(\bar{x},\bar{y}) = (a) \wedge (b) \wedge (c).$$

To the predicates (a)-(c) correspond three program segments, which can be expressed in the following form:

$$\{T\}\, S_1(\bar{x},\bar{y})\ \{(a)\}\, S_2(\bar{x},\bar{y}) \cdot \{(a)\wedge(b)\}\, S_3(\bar{x},\bar{y})\ \{(a)\wedge(b)\wedge(c)\}$$

We shall derive the program segment S_2:

$$\{(a)\}\, S_2(x,y)\ \{(a) \wedge \text{for } 1 \leqslant i \leqslant t',\ b_i' = b_i + (p'-1)(i-1)\}$$

The predicate (b) suggests the utilization of an iterative control structure with the following form

$$\{(a)\}\ \underline{\text{for}}\ k := 1\ \underline{\text{to}}\ t'\ \underline{\text{do}}\ S\ \{(a) \wedge (b)\}$$

The proof rule for the for statement can be expressed as follows [16]:

$$\frac{\{(a \leqslant k \leqslant b) \wedge P([a..k-1])\}\, S\, \{P([a..k])\}}{\{T\} \text{ for } k := a\ \underline{\text{to}}\ b\ \underline{\text{do}}\ S\ \{P([a.b])\}}$$

In the present case $P \equiv (b)$, hence through the application of the rule we get

$$\{(a) \wedge (1 \leqslant k \leqslant t') \wedge (b)\ [a..k-1]\ S\{(b)\ [a..k]\}$$

If we now apply the assignment axiom, we can state

$$S \equiv b_k := b_k + (p'+1)(i-1)$$

The program segment S_2 will then have the following form

```
{(a)} for k := 1 to t' do
        $b_k := b_k + (p'+1\ (i-1)$
          {(a) ∧ (b)}
```

The segment S_3 can be obtained in a similar manner. The derivation of S_1 will be the result of the successive application of the assignment axiom, leading to a result which is identical to the one produced for the first data representation (tuple of natural numbers). The expression of the cluster follows directly from what was said and is left as an exercise to the reader.

V. Conclusions

We have discussed the synthesis process of an Algol-like program by dealing separately with the algorithm and data aspects of the program. For the establishment of this separation we have used the concept of a cluster which is instrumental for providing a programming mechanism for the encoding of the data representation. The same effect could be obtained by mechanism such as classes [12] and forms [20].

In deriving a program statement we have proceeded through three distinct phases: derivation of a program schema from a formalized version of the problem definition; derivation of the problem data type ultimately in terms of formally well known and more primitive types (naturals and arrays in the given example); derivation of a programmed version of the data type definition (synthesis of the cluster). We not only defined the problem data type but also checked its correctness. The checking procedure differs from Hoare's [12] since we do not start from a completely defined type and a completely defined representation and try to define a mapping function connecting them.

We are presently trying to expand and formalize the concepts presented above through a simple example in search for a better understanding of the program derivation process. We are also investigating the idea of dealing with the problem of program transformations (in Gerhart's sense [9]) viewing these transformations along the two axes dealt with in this paper: algorithm and data.

References

[1] R. W. Floyd, "Assigning meanings to programs," in *Mathematical Aspects of Computer Science*, vol. XIX, V. T. Schwartz, Ed. Providence, RI: American Mathematical Society, 1967.

[2] R. M. Burstall and J. Darlington, "A transformation system for developing recursive programs," *J. Assoc. Comput. Mach.*, Jan. 1977.

[3] F. L. Bauer, "Programming as an evolutionary process," Institut Für Informatik, Technische Univ., Munchen, Berisht Nr. 7617, 1976.

[4] J. Arsac, *Nouvelles Lessons de Programmacion.* Paris, France: Dunod, to be published.

[5] E. Dijkstra, "Guarded commands, non-determinacy and a calculus for the derivation of programs," in *Proc. Int. Conf. on Reliable Software*, 1975.

[6] —, *A Discipline of Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1976.

[7] Z. Manna and R. Waldinger, "Synthesis: Dreams ⇒ programs," SRI Int., Tech. Note 156, Nov. 1977.

[8] S. L. Gerhart, "Knowledge about programs: A model and a case study," in *Proc. Int. Conf. Reliable Software*, Los Angeles, CA, 1975.

[9] —, "Proof theory of partial correctness verification systems," *SIAM J. Comput.*, vol. 5, no. 3, 1976.

[10] D. Gries, "An illustration of current ideas on the derivation of correctness proofs and correct programs," *IEEE Trans. Software Eng.*, vol. SE-2, Nov. 1976.

[11] N. Wirth, *Algorithms + Data Structure = Programs.* Englewood Cliffs, NJ: Prentice-Hall, 1976.

[12] C. A. R. Hoare, "Proof of correctness of data representations," *Acta Inform.*, vol. 1, 1972.

[13] B. Liskov and S. Zilles, "Programming with abstract data types," in *Proc. ACM SIGPLAN Conf. Very High Level Languages*, 1974.

[14] J. Dennis, "An example of programming with abstract data types," *Sigplan Notices*, vol. 10, no. 7, 1975.

[15] W. Wulf, R. L. London, and M. Shaw, "An introduction to the construction and verification of Alphard programs," *IEEE Trans. Software Eng.*, vol. SE-2, no. 4, 1976.

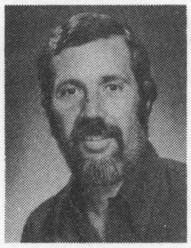
[16] C. A. R. Hoare and N. Wirth, "An axiomatic definition of the programming language Pascal," *Acta Inform.*, vol. 2, 1973.

[17] B. Liskov and S. Zilles, "Specification techniques for data abstractions," *IEEE Trans. Software Eng.*, vol. SE-1, Mar. 1975.

[18] J. Guttag, "Abstract data types and the development of data structures," *Commun. Assoc. Comput. Mach.*, vol. 20, no. 6, 1977.

[19] O. Dahl and C. A. R. Hoare, "Hierarchical program structures," in *Structured Programming.* New York: Academic Press, 1972.

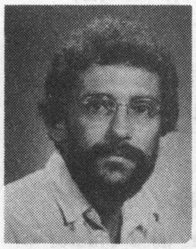
[20] W. A. Wulf, "Alphard: Toward a language to support structured programs," *Carnegie-Mellon Univ.*, Pittsburgh, PA, Tech. Rep., 1974.



Carlos J. P. Lucena (M'73) was born in Recife, Brazil. He received the B.S. degree from the Pontificia Universidade Catolica do Rio de Janeiro (PUC-RJ), Rio de Janeiro, Brazil, in 1965, the M.Math. degree from the University of Waterloo, Waterloo, Ont., Canada, in 1969 and the Ph.D. degree in computer science from the University of California at Los Angeles, in 1974.

Since 1965 he has been a faculty member of PUC-RJ where he is presently an Associate Professor and the Chairman of the Department of Computer Science. His main interests include programming methodology and design of programming languages.

Dr. Lucena is a member of the Brazilian Mathematical Society (SBM), the Brazilian Computing Society (SBC), the Association for Computing Machinery, and the Sigma Xi Society of American Scientists. He has been awarded a Guggenheim Fellowship for studies in programming systems and is presently serving as Technical Committee Chairman for the area of theoretical foundations for IFIP-80.



Tarcisio H. C. Pequeno was born in Fortaleza, Brazil. He received the B.S. degree in engineering from the Universidade Federal do Ceara (UFC), Ceara, Brazil, in 1970 and the M.S. degree in computer science from the Pontificia Universidade Catolica do Rio de Janeiro (PUC-RJ), Rio de Janeiro, Brazil, in 1977.

Since 1972 he has been an Assistant Professor at UFC. He is presently a Ph.D. candidate of PUC-RJ where he is a lecturer at the Department of Computer Science, on leave from UFC. His interests include theoretical foundations of programming.

Mr. Pequeno is a member of the Brazilian Computing Society (SBC) and a student member of the Association for Computing Machinery.