On the Development of Correct Specified Programs

ANDRZEJ J. BLIKLE

Abstract—The paper describes a method of program development which guarantees correctness. Our programs consist of an operational part, called instruction, and a specification. Both these parts are subject to the development and the refinement process. The specification consists of a pre- and postcondition called global specification and a set of assertions called local specification. A specified program is called correct if: 1) the operational part is totally correct w.r.t. the pre- and postcondition, 2) the precondition guarantees nonabortion, 3) local assertions are adequate for the proof of 1) and 2). The requirement of nonabortion leads to the use of a three-valued predicate calculus. We use McCarthy's calculus in that place. The paper contains a description of an experimental programming language PROMET-1 designed for our style of programming. The method is illustrated by the derivation of a bubblesort procedure.

Index Terms—Assertion-specified programs, bubblesort procedures, program correctness, program development, PROMET-1, sorting.

I. Introduction

THE PROBLEM of program correctness is frequently understood in a too narrow sense as the problem of proving programs correct. It is implicit in this understanding that program development and program verification are two independent processes, the first of which must be completed before the second starts. The scheme "first develop then prove" corresponds, maybe, to the way of establishing simple mathematical theorems, but is certainly inadequate for the use of mathematics in engineering. Nobody would dare to suggest that a civil engineer postpone the calculations until his bridge has been constructed. Why then is a software engineer supposed to be an exception?

The present paper describes a method of programming where program correctness is systematically controlled during program development. Of course, in order to talk about correctness one must have a standard against which to measure this correctness, i.e., a specification, and a standard of how to measure this correctness, i.e., a satisfiability relation. In our method the specification consists of a precondition, a postcondition, and a set of local assertions. Strictly speaking we are dealing here with specified programs of the form

$$\textbf{pre } c_{pr}\ IN \textbf{ post } c_{po}$$

where c_{pr} and c_{po} are the precondition and the postcondition, respectively, and where IN is an instruction with nested assertions. Such a program is called correct if: 1) IN is totally correct w.r.t. c_{pr} and c_{po}, i.e., c_{pr} guarantees nonlooping and c_{po} is satisfied upon termination; 2) c_{pr} guarantees nonabortion; 3) the assertions of IN are adequate for the proof of 1) and 2).

It is essential in our method that in program development we construct and refine both the virtual program and the specification. All development rules must be sound, i.e., must preserve program correctness. At the same time they may quite substantially change program meaning. This is in contrast to some other methods of program development where programs are developed from, rather than with, specifications [2], [5], [6], [11], [12]. The reason why we do not follow this style is twofold. First, in program development and maintenance one frequently has to change the specification. Programming is a creative art and any specification which is given ahead may turn out inadequate or at least incomplete. Second, the development of programs from specifications requires the concept of an equivalence relation between programs. This leads to tedious technical problems, since any practically acceptable equivalence is not a congruence [6].

Another issue which is substantial in our approach is the three-valued predicate calculus of McCarthy [21] used as a pattern in the definition of semantics of Boolean expressions. Except the clasical truth values *true* and *false* we admit the third value *undefined* which we need for an adequate treatment of abortion. Our rules of the evaluation of Boolean expressions are similar to that of many existing languages, e.g., of Pascal [17].

In the present paper we describe an experimental, simplified programming language PROMET-1 oriented towards the systematic development of correct specified programs. We start in Section II with the concept of an abstract data type which is fundamental for our style of programming. This is followed in Sections III and IV by the syntax and the denotational semantics of PROMET-1. Section V explains our two concepts of correctness: the global correctness of instructions and the correctness of specified programs. Program construction and modification rules are described in Section VI. In Section VII we give a detailed example of the derivation of a bubblesort procedure.

The idea of systematic program development is, of course, not new. Starting from Dijkstra [13] and Wirth [24] it has been advocated for many years and by many authors. In the recent few years it started to evolve towards a more disciplined

Manuscript received December 11, 1980; revised March 10, 1981. This work was supported in part by PAS MFCS Grant MR.I/3-04.1.1 and the National Science Foundation under Grant MCS 77-09906. Early versions of this paper were presented at the International Conference on Formal Methods and Mathematical Tools for the Construction of Software, Oberwolfach, West Germany, January 1979, the International Conference on Mathematical Foundations of Computer Science, Olomouc, September 1979, and the Fourth International Conference on Software Engineering, Munich, West Germany, September 1979.

The author is with the Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland.

approach and became the subject of theoretical research. Some references to this field were given above. Below we mention three other approaches which seem close to our method.

1) Bär [1] describes a method of program development in a modified Pascal. Programs are specified in a way very similar to ours, but program correctness does not cover nonabortion and must be proved in each step of program development.

2) Bjørner and others [3] develop a method based on the denotational approach to software specification. This approach is less formalized but has been used in serious applications including the design and/or description of operating systems, databases, and compilers.

3) Lee, de Roever, and Gerhard [18] describe a method similar to ours but restricted to the partial correctness of programs.

The present paper concludes the author's earlier research described partly in [5]-[9].

II. Abstract Data Types

The abstractness of data types is one of the basic principles of our style of programming. The user of PROMET is free in in the choice of his data type and can modify this data type along with the modification of the program. In this paper we omit the problem of data-type specification. We only require that data types are described as many-sorted algebras and that they satisfy a few technical assumptions. In the applications we allow any mathematical technique for the description of concrete many-sorted algebras. We also allow that some sorts and/or operations in data types are not implementable. Such nonimplementable objects, e.g., abstract reals and operations on them, are used at the level of program specifications.

By an *abstract data type* we mean a many-sorted algebra [14]

$$DT = (\{D_k\}_{k \in K}, \{f_\sigma\}_{\sigma \in \Sigma}, sort, arity)$$

where K is a set of elements called *sorts* (e.g., *real*, *integer*, *Boolean*, etc.), Σ is a set of elements called *functional symbols* (e.g., $+$, $-$, $\sqrt{\ }$, $\vee$, &, etc.), and *sort* and *arity* are functions which associate to every symbol in Σ the sort of its value and its arity, respectively. Formally, $sort: \Sigma \to K$ and $arity: \Sigma \to K^*$. For every sort k, D_k denotes the set of all elements of sort k. If for some $\sigma \in \Sigma$, $sort(\sigma) = k$ and $arity(\sigma) = k_1 \cdots k_n$, then we write $\sigma: k_1 \times \cdots \times k_n \to k$ and assume that f_σ is a total function which maps $D_{k_1} \times \cdots \times D_{k_n}$ into D_k. For instance, $\geqslant: real \times real \to Boolean$. If $arity(\sigma) = \epsilon$, where ϵ denotes the empty string, then we write $\sigma: \to k$. For instance, $1: \to integer$. In applications we identify every symbol σ with its meanings f_σ.

In order to provide the framework for carrying out the proofs of nonabortion, we assume that every D_k contains an abstract error $\perp_k$ [15]. In particular $D_{Boolean} = \{true, false, \perp_b\}$ and the logical connectives are defined in McCarty's [21] style (we return to this issue in Sections IV and VI):

$$p \to q, r = \begin{cases} q & \text{if } p = true \\ r & \text{if } p = false \\ \perp_b & \text{if } p = \perp_b \end{cases}$$

$$p \vee q = p \to true, q$$
$$p \,\&\, q = p \to q, false$$
$$p \supset q = p \to q, true$$
$$\sim p = p \to false, true.$$

We also assume that with every data type there is associated a family $\text{WFS} = \{(W_p, <_p)\}_{p \in P}$ of *well-founded sets*. For every $p \in P$, $<_p$ is a binary relation in W_p such that there is no infinite decreasing sequence $w_1 >_p w_2 >_p \cdots$ in W_p. Every relation $<_p$ has an appropriate symbol in Σ.

The last requirement about data types is that the set K contains the sort *any* such that $D_{any} = \cup\{D_k \mid k \in K - \{any\}\}$ and that with every sort k there is associated a *sort predicate* k: $any \to Boolean$ such that $f_k(d) = \textbf{if } d \in D_k \textbf{ then } true \textbf{ else } false$. For instance *integer* x is *true* iff x is an integer. Sorts and the corresponding sort predicates will be denoted by the same symbols.

III. PROMET—A Language for the Development of Specified Programs

PROMET stands for PROgramming METhod. Here we describe the first experimental version of this language called PROMET-1. As was mentioned already, PROMET-1 contains no mechanism for the specification of data types. We assume therefore that data types are defined outside of the language and that each concrete data type identifies an instance of PROMET-1. In other words, once we fix a data type we fix the corresponding syntactic and semantic primitives of the language. The syntax of PROMET-1 is restricted to iterative sequential programs with only global variables and without **goto**'s.

Let there be given a set of identifiers *IDE*, an abstract data type $DT = (\{D_k\}_{k \in K}, \{f_\sigma\}_{\sigma \in \Sigma}, sort, arity)$ and a family of well-founded sets $\text{WFS} = \{(W_p, <_p)\}_{p \in P}$. We define five syntactic classes: *EXP*—of *expressions*, *CON*—of *conditions*, *UIN*—of *unspecified instructions*, *INS*—of *instructions*, and *ASP*—of *assertion specified programs* (abbreviated *a.s. programs*).

$$EXP ::= IDE \mid CON \mid \sigma(\{EXP\}) \quad \text{for all} \quad \sigma \in \Sigma$$
$$\mid \textbf{if } CON \textbf{ then } EXP \textbf{ else } EXP \textbf{ fi}$$

$$CON ::= \sigma(\{EXP\}) \quad \text{for all} \quad \sigma \in \Sigma \quad \text{with } sort(\sigma) = Boolean$$
$$\mid \textbf{if } CON \textbf{ then } CON \textbf{ else } CON \textbf{ fi}$$
$$\mid (\forall\, IDE)CON \mid (\exists\, IDE)CON$$

Of course, $\{EXP\}$ denotes an arbitrary string of expressions. The choice of sorts in this string is delegated to semantics. We assume that in the applications we identify σ with f_σ and allow for infix notation. Quantified conditions are necessary for specifications.

$$UIN ::= \textbf{abort} \mid \textbf{skip} \mid \textbf{if } CON \textbf{ fi} \mid \{IDE\} := \{EXP\}$$
$$\mid UIN;\ UIN \mid \textbf{if } CON \textbf{ then } UIN \textbf{ else } UIN \textbf{ fi}$$

The string of identifiers on the left side of := is nonempty, repetition free, and of the same length as the string of expressions on the right side of :=. Instructions of the form **if** *CON* **fi** are called *tests*.

```
INS ::= UIN | INS as CON sa INS
        | if CON then INS else INS fi
        | while CON
            as CON ter EXP with IDE in p
            do INS od
        | inv CON INS vni
```

Conditions which appear after the key words **as** or **inv** are called *assertions.* Expressions which appear after **ter** in **while** instructions are called *termination expressions.* The symbol p is an element of P and is a name of a well-founded set. The assertion in the last instruction is called *permanent invariant* and the instruction themselve is called an *instruction with a declared invariant.* The key words **inv** and **vni** define the *syntactic scope* of the invariant. Assertions and termination expressions constitute local specifications and are skipped in program execution. Their role becomes essential in the definition of correctness (Section V):

ASP ::= **pre** CON INS **post** CON.

The conditions which follow **pre** and **post** are called, respectively, the *precondition* and the *postcondition.* In contrast to instructions, which represent algorithms, a.s. programs represent statements about algorithms.

IV. The Semantics of Expressions, Conditions, and Instructions

Here we give a routine denotational semantics of PROMET-1. Since our language contains neither **goto**'s nor procedures the denotational technique used below is very simple and may be understood by readers who are not familiar with it from elsewhere. The semantics of a.s. programs are postponed to Section V since it involves the concept of correctness.

To get started we need some notational conventions. For any sets A_1, A_2, A_3, any binary relations $R_1 \subseteq A_1 \times A_2$, $R_2 \subseteq A_2 \times A_3$, and any subset $C \subseteq A_2$, by $R_1R_2 = \{(a, b) | (\exists c)(aR_1c \,\&\, cR_2b)\}$ we denote the *composition* of R_1 and R_2 and by $R_1C = \{a | (\exists c)(aR_1c \,\&\, c \in C\}$ we denote the *coimage* of C w.r.t. R_1. For any A by I_A—or simply I if A is understood—we denote the *identity relation* in A, i.e., $I_A = \{(a, a) | a \in A\}$. By ϕ we denote the empty relation and the empty set. If $R \subseteq A \times A$ then $R^0 = I$ and $R^{i+1} = RR^i$ for any integer $i \geqslant 0$. By the *interation* of R we mean $R^* = \cup_{i=0}^{\infty} R^i$. For more details see [4]. By $[A_1 \to A_2]$ and $[A_1 \to A_2]_t$ we denote the sets of all partial, respectively, total functions from A_1 into A_2. By a *state* we mean a total function from the set of identifiers IDE into D_{any}. By $S = [IDE \to D_{any}]_t$ we denote the set of all states. The function of semantics is denoted by $[\ ]$ and maps syntactic objects into their denotations

$$[\ ] : EXP \to [S \to D_{any}]_t$$
$$[\ ] : CON \to [S \to \{true, false, \perp_b\}]_t$$
$$[\ ] : INS \to [S \to S]$$
$$[\ ] : ASP \to \{true, false\}.$$

Below we give the semantic clauses for $[\ ]$. Let x, E, and c, possibly with indexes, stand for identifiers, expressions, and conditions, respectively.

1) $[x](s) = s(x)$

2) $[\sigma(E_1, \cdots, E_n)](s)$

$$= \begin{cases} f_\sigma([E_1](s), \cdots, [E_n](s)) & \text{if the string of sorts of } [E_i](s) \text{ is compatible with the arity of } \sigma \\ \perp_k & \text{otherwise; where } k = sort(\sigma) \end{cases}$$

3) $[\textbf{if } c \textbf{ then } E_1 \textbf{ else } E_2 \textbf{ fi}](s)$

$$= \begin{cases} [E_1](s) & \text{if } [c](s) = true \\ [E_2](s) & \text{if } [c](s) = false \\ \perp_b & \text{if } [c](s) = \perp_b. \end{cases}$$

Let for any s, $s' \in S$ and $x \in IDE$, $s = s'$ *except in* x mean that $s(y) = s'(y)$, for any identifier y which is different from x. We do not request that $s(x) \neq s'(x)$, but we do not exclude this either.

4) $[(\forall x)c](s)$

$$= \begin{cases} true & \text{if } [c](s') = true \text{ for any } s' \text{ with } s' = s \text{ except in } x, \\ false & \text{if there exists } s' \text{ with } s' = s \text{ except in } x, \text{ such that } [c](s') = false \\ \perp_b & \text{otherwise, i.e., if for some } s', [c](s') = true \text{ and for the remaining } s' \text{ (but at least for one } [c](s') = \perp_b \end{cases}$$

5) $[(\exists x)c](s) = [\sim(\forall x)(\sim c)](s)$.

Observe that in 3) we introduce the mechanism of so-called *lazy evaluation*: in evaluating a conditional expression we do not evaluate both E_1 and E_2, but only this expression which is pointed to by the logical value of c. We return to this comment in Section VI. Now we define the semantics of INS. Let IN, possibly with indexes, denote instructions

6) $[\textbf{abort}] = \phi$

7) $[\textbf{skip}] = I_S$

8) $[\textbf{if } c \textbf{ fi}] = \{(s, s) | [c](s) = true\}$

9) $[x_1, \cdots, x_n := E_1, \cdots, E_n]$
$$= \{(s_1, s_2) | [E_i](s_1) \neq \perp \quad \text{for all } i \leqslant n \,\& \; s_2(x_i) = [E_i](s_1) \quad \text{for all } i \leqslant n \,\& \; s_2(y) = s_1(y) \quad \text{for all } y \notin \{x_1, \cdots, x_n\}\}$$

10) $[IN_1 ; IN_2] = [IN_1][IN_2]$

11) $[IN_1 \textbf{ as } c \textbf{ sa } IN_2] = [IN_1][IN_2]$

12) $[\textbf{if } c \textbf{ then } IN_1 \textbf{ else } IN_2 \textbf{ fi}]$
$$= [\textbf{if } c \textbf{ fi}]\,[IN_1] \cup [\textbf{if} \sim c \textbf{ fi}]\,[IN_2]$$

13) $[\textbf{while } c \textbf{ as } c_a \textbf{ ter } E \textbf{ with } x \textbf{ in } p \textbf{ do } IN \textbf{ od}]$
$$= ([\textbf{if } c \textbf{ fi}]\,[IN])^*\,[\textbf{if} \sim c \textbf{ fi}]$$

14) $[\textbf{inv } c\ IN \textbf{ vni}] = [IN]$.

Our semantics describe a very usual understanding of iterative instructions. All assertions are considered as comments. Their role becomes essential in the semantics of a.s. programs (Section V).

V. The Semantics of Assertion-Specified Programs

A.s. programs are correctness statements about instructions and therefore the function of semantics associates with them the truth values *true* and *false*. We shall say, however, that an a.s. program is *correct* or *incorrect* rather than *true* or *false*.

Intuitively, the correctness of **pre** c_{pr} IN **post** c_{po} means that the following two conditions are satisfied.

1) IN is *globally correct* w.r.t. c_{pr} and c_{po}, which means that for any state which satisfies c_{pr} the execution of IN terminates cleanly (neither loops nor aborts) and the terminal state satisfies c_{po}.

2) IN is *locally correct* w.r.t. c_{pr} and c_{po}, which means that the assertions nested in IN are adequate for the proof of 1).

Let for any $c \in CON$, $\{c\} = \{s \mid [c](s) = \text{true}\}$, i.e., $\{c\}$ is the set of all states which satisfy c. The property of global correctness is now easily formalized by the inclusion

$$\{c_{pr}\} \subseteq [IN]\{c_{po}\}.$$

It should be stressed that this property is stronger than the well-known total correctness [19]. The latter was defined under the assumption that all conditions and expressions represent total functions, i.e., that their evaluation can be always successfully completed. Since in real programming languages this assumption is never satisfied, all that total correctness means is the following. If c_{pr} is initially satisifed, then the execution of IN is finite and if it terminates cleanly (without error message) then c_{po} is satisfied upon termination. For instance, the program

```
pre integer array $A\,[0{:}n]$ & $a = A$ & $i = n$
   while $a[i] < a[i-1]$
      do $a[i], a[i-1] := a[i-1], a[i]$;
         $i := i - 1$ od
post $a$ is a permutation of $A$
```

can be proved totally correct, although it aborts whenever i reaches the value 0. Of course, this program is not globally correct.

The concept of global correctness was originally introduced (under another name) by Mazurkiewicz [20] for the case of abstract iterative processes. Global correctness proof rules for iterative programs were established later by Blikle [10] and were proved consistent and algebraically complete. Here these proof rules constitute cases in the definition of correctness of a.s. programs. Let for the convenience of notation the diagrams

$$\downarrow \frac{X}{Y} \quad \text{and} \quad \updownarrow \frac{X}{Y}$$

be read: *if X then Y* and *X iff Y*, respectively. Let for c_1, $c_2 \in CON$, $c_1 \Rightarrow c_2$ denote the fact that $\{c_1\} \subseteq \{c_2\}$ and be read as c_1 *implies* c_2. The correctness of a.s. programs is defined by structural induction. D stands for *definition*.

(D1) If UI is an unspecified instruction, then

$$\updownarrow \begin{array}{l} \textbf{pre } c_{pr}\ UI \textbf{ post } c_{po} \\ \hline \{c_{pr}\} \subseteq [UI]\{c_{po}\} \end{array}$$

For unspecified instructions correctness means, of course, global correctness.

(D2)
$$\updownarrow \begin{array}{l} \textbf{pre } c_{pr}\ IN_1 \textbf{ as } c_a \textbf{ sa } IN_2 \textbf{ post } c_{po} \\ \hline 1)\ \textbf{pre } c_{pr}\ IN_1 \textbf{ post } c_a \\ 2)\ \textbf{pre } c_a\ IN_2 \textbf{ post } c_{po} \end{array}$$

(D3)
$$\updownarrow \begin{array}{l} \textbf{pre } c_{pr} \textbf{ if } c \textbf{ then } IN_1 \textbf{ else } IN_2 \textbf{ fi post } c_{po} \\ \hline 1)\ c_{pr} \Rightarrow c \vee \sim c \\ 2)\ \textbf{pre } c_{pr}\ \&\ c\ \ IN_1 \textbf{ post } c_{po} \\ 3)\ \textbf{pre } c_{pr}\ \&\ \sim c\ IN_2 \textbf{ post } c_{po} \end{array}$$

Observe that in the two-valued logic formula 1) in (D3) is always satisfied. Here it means that whenever c_{pr} is true, the value of c is defined, i.e., the execution of c does not lead to abortion.

(D4)
$$\updownarrow \begin{array}{l} \textbf{pre } c_{pr} \textbf{ while } c \textbf{ as } c_a \textbf{ ter } E \textbf{ with } x \textbf{ in } p \textbf{ do } IN \textbf{ od post } c_{po} \\ \hline 1)\ c_{pr} \Rightarrow c_a \\ 2)\ c_a \Rightarrow c \vee \sim c \\ 3)\ \text{i) } [E] : \{c_a\} \to W_p \text{ is a total function,} \\ \quad\ \ \text{ii) in } IN\ x \text{ occurs at most in the assertions} \\ 4)\ \textbf{pre } c_a\ \&\ c\ \&\ E = x\ IN \textbf{ post } c_a\ \&\ E <_p x \\ 5)\ c_a\ \&\ \sim c \Rightarrow c_{po} \end{array}$$

Here c_a is called *loop assertion* and describes the environment in which our loop is executed. In this environment c has a defined value [c.f. 2)], the value of E is in the well-founded set W_p [cf. 3)], whenever c is satisfied then IN can be executed, keeps the state in the environment and decrements the value of E [cf. 4)], if $\sim c$ is satisfied, then c_{po} is satisfied [cf. 5)].

(D5)
$$\updownarrow \begin{array}{l} \textbf{pre } c_{pr} \textbf{ inv } c\ IN \textbf{ vni post } c_{po} \\ \hline \textbf{pre } c_{pr}\ IN\,(c_a\ \&\ c/c_a) \textbf{ post } c_{po} \end{array}$$

Here $IN\,(c_a\ \&\ c/c_a)$ denotes the instruction which results from IN by the replacement of any assertion c_a by $c_a\ \&\ c$. Intuitively, the invariant declaration is a way of the factorization of these conditions which could be added to all assertions. This construction has been introduced for syntactical convenience. If IN contains no assertions (is unspecified), then the invariant declaration is effectless.

As has been proved in [10], local correctness always implies global correctness, i.e.,

$$\downarrow \begin{array}{l} \textbf{pre } c_{pr}\ IN \textbf{ post } c_{po} \\ \hline \{c_{pr}\} \subseteq [IN]\{c_{po}\} \end{array}$$

In other words the global correctness proof rules (D1)-(D5) are consistent. It was also proved in [10] that every global-correctness assertion is a Floyd's assertion but not vice versa; some Floyd's assertions are too weak to be used in the proof of global correctness.

To conclude this section let us briefly explain the motivation for having assertions in our programs. First, assertions are necessary in program development. As will be seen later, many of our program modification rules strongly rely on programs' local properties. Second, they can be used in program testing. If only implementable, they must be satisfied in program execution. Third, they record programmer's arguments for the correctness of the program. Such arguments not only justifly the credibility of the program, but also may be useful in program exploitation and maintenance.

VI. Program Development Rules

It is not the purpose of this section to give a complete set of program development rules for PROMET-1. Different classes of algorithmic problems may require different development rules and the user of PROMET is expected to develop such rules by himself. Our method only provides a mathematical framework where these rules may be defined and proved sound. Of course, there is a small set of rules used in nearly every program development. Some of them are described below.

A program development rule is called *sound* if it preserves program correctness. In developing programs by sound rules we have to make sure that the initial program is correct and that each rule is applicable whenever used.

The first set of rules is implicit in the definition of correctness of Section V. Indeed, each of the diagrams (D1)-(D5) when read bottom-up is such a rule. These rules are called *basic construction rules.*

The next set of rules contains *condition modifications* (*CM*). In order to describe them we have to introduce some metarelations between conditions. Let $c_1, c_2, c_3 \in CON$ and let an abstract data type DT be fixed. We say that c_1 and c_2 are *strongly*, respectively, *weakly equivalent* (in DT), which we denote by

$$c_1 \eqsim c_2, \text{ respectively, } c_1 \Leftrightarrow c_2$$

if $[c_1] = [c_2]$, respectively, $\{c_1\} = \{c_2\}$. In the first case c_1 and c_2 represent the same logical (three-valued) function, in the second–whenever one is satisfied, then so is the other and vice versa, e.g., if n and x range over real numbers, then $x \geqslant 0 \,\&\, n > x^2 \Leftrightarrow x \geqslant 0 \,\&\, \sqrt{n} > x$, but the strong equivalence does not hold since for $n < 0$, $n > x^2$ is false and $\sqrt{n} > x$ is undefined. On the other hand, $n \geqslant 0 \,\&\, x \geqslant 0 \,\&\, n > x^2 \eqsim n \geqslant 0 \,\&\, x \geqslant 0 \,\&\, \sqrt{n} > x$.

As is easy to see both relations $\eqsim$ and $\Leftrightarrow$ are equivalences, but only the first one is a congruence. Indeed, $c_1 \Leftrightarrow c_2$ does not imply $\sim c_1 \Leftrightarrow \sim c_2$, e.g., $x^{-1} > 0 \,\&\, x = y - 1 \Leftrightarrow (y-1)^{-1} > 0 \,\&\, x = y - 1$, but $x^{-1} \leqslant 0 \vee x \neq y - 1 \not\Leftrightarrow (y-1)^{-1} \leqslant 0 \vee x \neq y - 1$. In the algebra of conditions with $\eqsim$ both & and $\vee$ are associative and distributive, and obey de Morgan's laws. Of course, they are not commutative. On the other hand, & is commutative with $\Leftrightarrow$, i.e., $c_1 \,\&\, c_2 \Leftrightarrow c_2 \,\&\, c_1$, but $\vee$ is not.

In the examples of weakly and strongly equivalent conditions we had the situation that two conditions were compared in the context of a third. For instance, $n > x^2$ and $\sqrt{n} > x$ were compared in the context of $x \geqslant 0$. Since this situation is typical in program development we introduce a more explicit notation. We write

$$\begin{array}{l}
\quad c_1 \eqsim c_2 \ \textit{whenever}\ c_3 \\
\text{if } c_3 \,\&\, c_1 \eqsim c_3 \,\&\, c_2 \text{ and} \\
\quad c_1 \Leftrightarrow c_2 \ \textit{whenever}\ c_3 \\
\text{if } c_3 \,\&\, c_1 \Leftrightarrow c_3 \,\&\, c_2 .
\end{array}$$

For instance,

$$\begin{array}{l}
n > x^2 \eqsim \sqrt{n} > x \ \textit{whenever}\ n \geqslant 0 \,\&\, x \geqslant 0 \\
n > x^2 \Leftrightarrow \sqrt{n} > x \ \textit{whenever}\ x \geqslant 0.
\end{array}$$

(CM.1) If in a correct a.s. program we replace

1) any **while-do** or **if-then-else** condition c_1 by c_2 such that $c_1 \eqsim c_2$, or

2) any precondition, postcondition, or assertion c_1 by c_2 such that $c_1 \Leftrightarrow c_2$, then the resulting program is correct. □

(CM.2) If in a correct a.s. program with a declared invariant c we replace any assertion c_1 which appears within the syntactic scope of the declaration of c by c_2 such that $c_1 \Leftrightarrow c_2$ *whenever* c, then the resulting program is correct. □

(CM.3)

$$\begin{array}{|l}
1)\ \textbf{pre}\ c_{pr}\ \textbf{while}\ c_1\ \textbf{as}\ c_a\ \textbf{ter}\ E\ \textbf{in}\ p\ \textbf{do}\ IN\ \textbf{od post}\ c_{po} \\
2)\ c_1 \eqsim c_2\ \textit{whenever}\ c_a \\
\hline
\downarrow \quad \textbf{pre}\ c_{pr}\ \textbf{while}\ c_2\ \textbf{as}\ c_a\ \textbf{ter}\ E\ \textbf{in}\ p\ \textbf{do}\ IN\ \textbf{od post}\ c_{po}
\end{array}$$

(CM.4)

$$\begin{array}{|l}
1)\ \textbf{pre}\ c_{pr}\ IN\ \textbf{post}\ c_{po} \\
2)\ c'_{pr} \Rightarrow c_{pr}\ \text{and}\ c_{po} \Rightarrow c'_{po} \\
\hline
\downarrow \quad \textbf{pre}\ c'_{pr}\ IN\ \textbf{post}\ c'_{po}.
\end{array}$$

Easy proofs of the soundness of these rules are omitted. We may add that by 2) of (CM.1) the connective & is commutative in preconditions, postconditions, and assertions.

The applications of CM-rules are rather obvious. Some are shown in Section VII some others in [9]. Below we show a typical application of (CM.3). Let $[\sqrt{n}]$ denote the integer square root, let *nnint* denote the WFS of nonnegative integers. The program

```
pre $n \geqslant 0 \,\&\, x = 0$
   while $x < [\sqrt{n}]$
      as $n, x \geqslant 0 \,\&\, x \leqslant [\sqrt{n}]$ ter $[\sqrt{n}] - x$ with $z$ in $nnint$
      do $x := x + 1$ od
post $n \geqslant 0 \,\&\, x = [\sqrt{n}]$                                    (1)
```

computes $[\sqrt{n}]$, but refers to this function in the **while** condition. This is a typical situation at an early stage of program development. We eliminate the unwanted condition on the strength of the conditional equivalence $x < [\sqrt{n}] \eqsim (x+1)^2 \leqslant n$ *whenever* $n, x \geqslant 0$ and get

```
pre $n \geqslant 0 \& x = 0$
   while $(x + 1)^2 \leqslant n$
      as $n, x \geqslant 0 \& x \leqslant [\sqrt{n}]$ ter $[\sqrt{n}] - x$ with $z$ in $nnint$
      do $x := x + 1$ od
post $n \geqslant 0 \& x = [\sqrt{n}]$.                                    (2)
```

The next class of rules consist of program enrichments (PE). The first rule supplies a new identifier y and extends the program in such a way that for a certain expression E the equation $y = E$ becomes a permanent invariant. Let $E(E'/x)$ denote the result of the substitution of E' for all quantifier-free occurrences of x in E. Let $IN|x = E$ denote the instruction which results in from IN by the replacement of every assignment $x_1, \cdots, x_n := E_1, \cdots, E_n$ in IN, such that at least one x_i appears in E, by the assignment $x_1, \cdots, x_n, y := E_1, \cdots, E_n, E(E_1/x_1, \cdots, E_n/x_n)$. Let a *c-computation* of IN be a sequence of states $s_1, s_2, \cdots$ generated during the execution of IN and such that s_1 satisfies the condition c. Of course all the three concepts may be easily formalized.

(PE.1)

1) **pre** c_{pr} IN **post** c_{po},
2) y does not appear in the program above,
3) the value of E is defined for any state of any c_{pr}-computation of IN

pre $c_{pr} \& y = E$ **inv** $y = E$ $IN | y = E$ **vni post** $c_{po} \& y = E$

A routine but tedious inductive proof of this rule is omitted. The rule has two major applications: the transformation of programs from one data type into another [7] and program optimization. An example of the latter is the introduction of y with $y = (x + 1)^2$ into the program 2). Since after such a transformation the permanent invariant $y = (x + 1)^2$ becomes a semantic part of the loop assertion, we can modify **while** condition on the strength of $(x + 1)^2 \leqslant n \approx y \leqslant n$ *whenever* $y = (x + 1)^2$ and (CM.3). We can also replace the assignment $x, y := x + 1, ((x + 1) + 1)^2$ by $x, y := x + 1, y + 2x + 3$. On the strength of $y = (x + 1)^2$ the latter is semantically equivalent to the former. As a result we get

```
pre $n \geqslant 0 \& x = 0 \& y = (x + 1)^2$
   inv $y = (x + 1)^2$
      while $y \leqslant n$
         as $n, x \geqslant 0 \& x \leqslant [\sqrt{n}]$ ter $[\sqrt{n}] - x$ with $z$ in $nnint$
         do $x, y := x + 1, y + 2x + 3$ od
   vni
post $n \geqslant 0 \& x = [\sqrt{n}] \& y = (x + 1)^2$.
```

The second PE-rule serves in the development of **while** loops. Many instances of such loops may be regarded as vehicles with two distinguished mechanisms: *step* and *keep*. The *step* mechanism proceeds from one element into another in a WFS; the *keep* mechanism is responsible for keeping a certain assertion c_a true in all steps. Since the *step* mechanism usually destroys the truth of c_a, the *keep* mechanism must provide a recovery. For instance, in the loop which calculates $\Sigma_{j=1}^{n} a_j$ and stores this value in x, *step* consists of $i := i + 1$, the assertion c_a is $x = \Sigma_{j=1}^{i} a_j$ and *keep* is $x := x + a_i$. The *step-and-keep* viewpoint suggests a certain strategy in developing loops. First develop a pure *step* mechanism, then enrich it by an appropriate *keep* mechanism. The following rule formalizes this strategy for **while-do** instructions.

(PE.2)

1) **pre** c_{pr} **while** c **as** c_a **ter** E **with** x **in** p **do** IN **od post** c_{po}
2) **pre** $c_a \& c \& c_1$ IN **post** $c_a \& c_1'$
3) *in* IN_1, x occurs at most in the assertions
4) **pre** $c_a \& c_1' \& E <_p x$ IN_1 **post** $c_a \& c_1 \& E <_p x$

```
pre $c_{pr} \& c_1$
   while $c$
      as $c_a \& c_1$ ter $E$ with $x$ in $p$
      do
         $IN$
         as $c_a \& c_1' \& E <_p x$ sa
         $IN_1$
      od
post $c_{po} \& c_1$
```

Here program 1) represents the step mechanism. We wish to modify it in such a way that the new loop preserves condition c_1. Since IN destroys this condition [cf. 2)] we add an instruction IN_1 which provides the recovery [cf. 4)]. We also assume that IN_1 does not increase too much the value of E. The application of this rule is shown in Section VII.

The proof of the soundness of (PE.2) consists of showing that the resulting program is correct in the sense of (D4). Cases 1), 2), 3), and 5) of (D4) are obvious. In order to show 4), observe that by the assumption 1) of (PE.2) **pre** $c_a \& c \& E = x$ IN **post** $c_a \& E <_p x$. On the other hand, IN transforms c_1 into c_1', hence (formally this step requires a new CM-rule) **pre** $c_a \& c \& c_1 \& E = x$ IN **post** $c_a \& c_1' \& E <_p x$. This, together with the assumption 4) of (PE.2) and by the rule (D2) leads to the required conclusion: **pre** $c_a \& c_1 \& c \& E = x$ IN **as** $c_a \& c_1' \& E <_p x$ **sa** IN_1 **post** $c_a \& c_1 \& E <_p x$.

The last rule which we describe here belongs to the group of *program truncations* and consists of the removal of an identifier from a program. This rule is used in space optimization and in transforming programs from one data type into another. An identifier x is called *autonomous* in an instruction IN if it appears at most in assertions and in the assignments which modify x. By IN/x we denote the instruction which results from IN by the removal of all assignments of the form $x := E$ and by the existential quantification of all free accurences of x in the assertions of IN.

(PT.1)

1) **pre** c_{pr} IN **post** c_{po}
2) x is autonomous in IN

pre $(\exists x)c_{pr}$ IN/x **post** $(\exists x)c_{po}$

The quantification of x in IN requires special attention if x

occurs simultaneously in a declared invariant and in an assertion which is in the syntactic scope of this invariant. In such a case we cannot quantify x separately in the invariant and in assertions. We have to eliminate x either from the invariant or from the assertions and if this is impossible we have to move the invariant down to all corresponding assertions. For applications of (PT.1), see [7] and [9].

VII. An Example of Program Derivation: Bubblesort

To get started we recall the intuitive idea of bubblesort. Suppose that we are given a vertical column of bubbles, each bubble having a certain weight. Suppose that our bubbles are immersed in an environment which satisfies the following Archimedes principle: each bubble which is ligher than its upper neighbor tends to swap with this neighbor in moving up. At some initial moment all the bubbles are glued together which prevents them from swapping. In the first step of bubblesort we free the first bubble from the top. Of course, nothing will happen since this bubble has no upper neighbor. Next we free the second bubble. This time a swap may occur if the second bubble is lighter than the first one. In each successive step of our procedure we free the successive bubble which immediately starts to move up in searching for such a position in the column which does not violate the Archimedes principle. It is intuitively clear that in the last step of our procedure the column of bubbles will be ordered according to the increased weights.

In developing programs in PROMET the first step always consists of the description of the appropriate data type. Then we write the first version—or approximation—of our program and from now on we develop and modify both the data type and the program. Since PROMET-1 is not equipped with the formalism for data specification, we shall use here, in that place, the language of intuitive mathematics. In our example the initial data type is the following.

Sorts:

Int-integers;

Arr-arrays; each array is a total function

$a : \{0, \cdots, n\} \to Int$, where $n \geqslant 0$;

Bol-$\{true, false, \bot_b\}$.

Functions (with Non-Boolean Values):

+, -, 0, 1-the arithmetical functions and constants understood in the standard way;

maxind: $Arr \to Int$-the maximal index of an array

component: $Arr \times Int \to Int$-the ith component of an array; according to the common style we shall write $a[i]$ for $component(a, i)$

seg: $Arr \times Int \to Arr$-the initial segment: $seg(a, j) = (a(0), \cdots, a(j))$ for $0 \leqslant j \leqslant maxind\ a$; according to the common style we shall write $a[0:j]$ for $seg(a, j)$.

Predicates:

integer, *array*-the sort predicates (Section II)

$\leqslant$, $<$-the usual arithmetical inequalities

is sorted: $Arr \to Bol$

a *is sorted* $:\approx (\forall\ integer\ i)$

$(0 \leqslant i < maxind\ a \supset a(i) \leqslant a(i+1))$

perm: $Arr \times Arr \to Bol$

a_1 *perm* $a_2 :\approx a_1$ is a permutation of a_2.

WFS:

nnint = $(\{0, 1, 2, \cdots\}, <)$-nonnegative integers.

Having defined our data type we can proceed to the development of the program. Following the physical model of bubblesort, we shall assume that the program consists of two mechanisms, as follows:

1) the *step* mechanism, which moves a certain pointer j top-down along the array; intuitively this pointer releases successive bubbles,

2) the *keep* mechanism, which guarantees that in every step the segment $a[0:j]$ has been sorted.

In order to describe these mechanisms we first establish a condition which characterizes the invariant properties of identifiers in the program. This condition will be later declared as a permanent invariant and will be referred to as *environment*-1:

array A, a & a *perm* A &
integer k, j & k = *maxind* A & $0 \leqslant j \leqslant k$.

Here A is a (constant) external array and a is its internal copy. The integer j represents a pointer which moves from 0 to k. Now we can describe the step mechanism of our loop.

```
pre environment-1 & $a = A$ & $j = 0$
   inv environment-1
      while $j < k$
            as true ter $k - j$ with $x_1$ in nnint
            do $j := j + 1$ od
   vni
post environment-1 & $j = k$.                                  (3)
```

This program is obviously correct. The keep mechanism of our prospective loop should keep the truth of the assertion $a[0, j]$ *is sorted.* In order to introduce it into (3) we apply (PE.2). First observe that the following program is correct [cf. 2), (PE.2)]:

```
pre environment-1 & $a[0:j]$ is sorted & $j < k$
   $j := j + 1$
post environment-1 & $j > 0$ & $a[0:j-1]$ is sorted.           (4)
```

Our recovery instruction—call it <BUBBLING>—must satisfy 3) and 4) of (PE.2). This means that it must not change the terminal variable x_1 and must make the following a.s. program correct:

```
pre environment-1 & $j > 0$ & $a[0:j-1]$
                                    is sorted & $k - j < x_1$
   inv environment-1
      <BUBBLING>
   vni
post environment-1 & $a[0:j]$ is sorted & $k - j < x_1$.
                                                               (5)
```

According to (PE.2), for any instruction <BUBBLING> which does not modify x_1 and satisfies (5) the following program is correct:

```
pre environment-1 & a = A & j = 0
   inv environment-1
      while j < k
         as a[0:j] is sorted ter k - j with x_1 in nnint
         do j := j + 1
            as j > 0 & a[0:j - 1] is sorted & k - j < x_1 sa
            <BUBBLING>
         od
   vni
post environment-1 & j = k & a[0:j] is sorted.                (6)
```

This is a scheme of a sorting procedure where we have to substitute an appropriate instruction for <BUBBLING>. In order to design this instruction we extend our data type by adding one new sort, two new functions, and one new predicate.

Sort:

Vec-vectors; each vector is a total function $v : N \to Int$ where N is an arbitrary finite set of integers

Functions:

swap: Arr × *Int* × *Int* → *Arr*; *swap*(a, i, j) is, for $0 \leqslant i, j \leqslant$ *maxind a*, the result of swapping the ith with the jth element in a

but: Arr × *Int* → *Vec*; *a but i* is, for $0 \leqslant i \leqslant$ *maxind a*, the restriction of array a to the domain $\{0, \cdots, \textit{maxind } a\} - \{i\}$

Predicates:

bubbles in: Int × *Arr* → *Bol*; *i bubbles in a* :≈
$0 \leqslant i \leqslant$ *maxind a*
& *a but i is sorted* & $i <$ *maxind a*
$\supset a[i + 1] \geqslant a[i]$.

It is understood that the predicate *is sorted* has been extended to the sort of vectors. Our instruction <BUBBLING> will be a loop where the step mechanism moves a new pointer i (which simulates the ascending bubble) and the keep mechanism keeps true the assertion *i bubbles in* $a[0, j]$. In order to describe this loop we have to extend our environment by adding new identifier i. Let then *environment*-2 denote

environment-1 & *integer i* & $0 \leqslant i \leqslant j$.

Observe that in all former programs we may replace *environment*-1 by *environment*-2 and these programs remain correct. First approximation of <BUBBLING> may be written offhand

```
pre environment-2 & j > 0 & a[0:j - 1] is sorted
                                        & k - j < x_1 & i = j
inv environment-2
   while ~a[0:j] is sorted
      as i bubbles in a[0:j] & k - j < x_1
                                    ter i with x_2 in nnint
      do i := i - 1;
         a := swap(a, i, i + 1)
      od
vni
post environment-2 & a[0:j] is sorted & k - j < x_1.

                                                          (7)
```

This program is correct, which is easy to prove, but its while condition is hardly implementable. We replace this condition by $i > 0$ & $a[i] < a[i - 1]$ in applying (CM.3) and the conditional equivalence

~$a[0:j]$ *is sorted* ≈ $i > 0$ & $a[i] < a[i - 1]$
whenever
environment-2 & *i-bubbles in* $a[0:j]$ & $k - j < x_1$.

Observe that in this transformation we strongly rely on the local specification of our program. The existence of this specification allows us for the immediate application of (CM.3).

After having applied this transformation we prefix (7) with $i := j$ and get the final version of <BUBBLING>. We substitute this instruction into (6) replacing at the same time $j = k$ & $a[0:j]$ *is sorted* by $j = k$ & *a is sorted* in the postcondition. The following is the final bubblesort program:

```
pre environment-2 & a = A & j = 0
   inv environment-2
      while j < k
         as a[0:j] is sorted ter k - j with x_1 in nnint
         do j := j + 1
            as j > 0 & a[0:j - 1] is sorted & k - j < x_1 sa
            i := j
            as j > 0 & i = j & a[0:j - 1] is sorted
                                              & k - j < x_1 sa
            while i > 0 & a[i] < a[i - 1]
               as i bubbles in a[0:j] & k - j < x_1
                                           ter i with x_2 in nnint
               do
                  i := i - 1;
                  a := swap (a, i, i + 1)
               od
         od
   vni
post environment-2 & j = k & a is sorted.
```

VIII. Final Remarks

The experiences collected with PROMET-1 (see Acknowledgment) proved that it may be an adequate tool in developing programs of a textbook size. Of course, in order to become a more realistic language, PROMET must be equipped with additional mechanisms such as type declaration, blocks, procedures, etc. This work is in progress now.

A routine question raised in connection with any formal method of programming is to what extent should such a method be applied formally. In the author's opinion the appropriate level of formality is that of intuitive mathematics. Evident need not be proved, but everything must be provable. Whether to carry out a formal (but never formalized) proof is left to the programmer.

Acknowledgment

The author wishes to thank H. E. Sengler whose discovery of a mistake in an early version of this paper gave rise to the introduction of the three-valued logic into PROMET, and to A. Mazurkiewicz who suggested McCarthy's logic. He is also grateful to K. Apt and B. M. Reynaud for their stimulating remarks. The ideas of this paper were included in three courses which the author gave in Berkeley (1979), in Warsaw (1979), and in Lyngby (1980). The discussions which he had in his classes certainly influenced the approach. He also wishes to thank in particular D. Brotsky from Berkeley, A. Tarlecki from Warsaw, and N. Anderson from Lyngby.

References

[1] R. Bär, "A methodology for simultaneously developing and verifying Pascal programs," in *Proc. Constructing Quality Software, IFIP TC-2 Working Conf.*, Novosybirsk, May 1977. Amsterdam, The Netherlands: North-Holland, 1978.

[2] F. L. Bauer *et al.*, "Towards a wide spectrum language to support program specification and program development," *SIGPLAN Notices*, Dec. 1978.

[3] D. Bjørner, "The Vienna development method (VDM): Software specification and program synthesis," in *Proc. Int. Conf. Math. Studies Inform. Processing*, K. E. Blum, M. Paul, and S. Takasu, Eds., Kyoto, Japan, Aug. 1978. New York: Springer, 1979.

[4] A. Blikle, "A comparative review of some program-verification methods," in *Proc. 6th. Symp. Math. Foundations Comput. Sci.*, Tatranska Łomnica, 1977; also in *Lecture Notes in Computer Science*, vol. 53. Heidelberg: Springer, 1977, pp. 17-33.

[5] —, "A mathematical approach to the derivation of correct programs," in *Proc. Int. Workshop Semantics Programming Lang.*, Bad Honnef, Mar. 1977; also in Abteilung Inform., Univ. Dortmund, Bericht Nr 4.1., 1977, pp. 25-29.

[6] —, "Towards mathematical structured programming," in *Proc. IFIP Working Conf. Formal Description Programming Concepts*, St. Andrews, N.B., Canada, Aug. 1-5, 1977, E. J. Neuhold, Ed. Amsterdam, The Netherlands: North-Holland, 1978.

[7] —, "Specified programming," in *Proc. Int. Conf. Math. Studies Inform. Processing*, K. E. Blum, M. Paul, and S. Takasu, Eds., Kyoto, Japan, Aug. 1978. New York: Springer, 1979.

[8] —, "Assertion programming," in *Proc. Int. Conf. Math. Foundations Comput. Sci.*, J. Becvar, Ed. Heidelberg: Springer, 1979.

[9] —, "On correct program development," in *Proc. 4th Int. Conf. Software Eng.*, Munich, Sept. 1979, pp. 164-173, IEEE Cat. 79CH1479-5C.

[10] —, "The clean termination of iterative programs," *Acta Informatica*, to be published.

[11] M. Broy *et al.*, "Methodical solution of the problem of ascending subsequences of maximum length within a given sequence," *Inform. Process. Lett.*, vol. 8, no. 5, pp. 224-229, 1979.

[12] R. M. Burstal and J. Darlington, "A transformation system for developing recursive programs," *J. Ass. Comput. Mach.*, vol. 24, pp. 44-67, 1977.

[13] O. J. Dahl, E. W. Dijkstra, and C.A.R. Hoare, *Structured Programming*. New York: Academic, 1972.

[14] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, "Abstract data types as initial algebras and correctness of data representations," in *Proc. Conf. Comput. Graphics, Pattern Recog. Data Structure*, May 1975, pp. 89-93.

[15] —, "Abstract errors for abstract data types," in *Proc. IFIP Working Conf. Formal Description Programming Concepts*, E. Neuhold, Ed. New York: North-Holland, 1978.

[16] M.J.C. Gordon, *The Denotational Description of Programming Languages*. New York: Springer, 1979.

[17] K. Jensen and N. Wirth, *Pascal User Manual*, 2nd ed. New York: Springer, 1975.

[18] S. Lee, W. P. de Roever, and S. L. Gerhart, "The evolution of list-copying algorithms," in *Proc. 6th ACM Symp. Principles Programming Lang.*, Jan. 1979.

[19] Z. Manna and A. Pnueli, "Axiomatic approach to total correctness of programs," *Acta Informatica*, 1974.

[20] A. Mazurkiewicz, "Proving properties of processes," an invited lecture at MFCS, Strbske Pleso, 1973; also in *Algorytmy*, vol. 11, pp. 5-22, 1974.

[21] J. McCarthy, "A basis for a mathematical theory of computation," presented at Western Joint Comput. Conf., May 1961; also in *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg, Eds. Amsterdam, The Netherlands, North-Holland, 1967, pp. 33-70.

[22] C. Pair, "La construction des programmes, R.A.I.R.O." *Informatique*, vol. 13, pp. 113-137, 1979.

[23] H. Rasiowa, *An Algebraic Approach to Non-Clasical Logic*. Amsterdam, The Netherlands: North-Holland, 1974.

[24] N. Wirth, "Program development by stepwise refinement," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 221-227, 1971.

Andrzej Blikle received the M.S. degree in mathematics from Warsaw University, Warsaw, Poland, in 1962, and the Ph.D. degree in mathematics from the Polish Academy of Sciences, Warsaw, Poland, in 1966.

In 1964 he joined the Mathematical Institute of the Polish Academy of Sciences. Since 1971 he has been with the Institute of Computer Science of the Polish Academy of Sciences, where he is currently a Professor. He has also been a Visiting Professor at the University of Waterloo, Waterloo, Canada, the University of California, Berkeley, the Technical University of Denmark, Lyngby, and Linköping University, Linköping, Sweden. His professional interests center around program correctness and the mathematical semantics of programming languages.

Dr. Blikle is a member of the Polish Mathematical Society, the Polish Computer Science Society, the American Mathematical Society, and the Association for Symbolic Logic.