ON THE DEVELOPMENT OF CORRECT PROGRAMS WITH THE DOCUMENTATION

by

Andrzej Blikle

Memorandum No. UCB/ERL M79/25

23 April, 1979

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94704

ON THE DEVELOPMENT OF CORRECT PROGRAMS WITH THE DOCUMENTATION

by

Andrzej Blikle†

Institute of Computer Science

Polish Academy of Sciences

PKiN, P.O. Box 22

Phone: 20-38-88 Telex 813556

Key words and phrases: Systematic program development, total correctness, non-abortion, sound transformations, McCarthy's logic, documentation by assertions

CR Catagories: 5.24, 5.21

---

†Currently visiting the Department of Electrical Engineering and Computer Sciences, Computer Science Division, University of California, Berkeley, California, 94720

Research supported in part by the National Science Foundation Grant MCS77-09906

An earlier version of this paper was presented at the international conference Formale Methoden un Mathematische Hilfsmittel für die Softwarekonstruktion, Oberwolfach, January 3-13, 1979.

ABSTRACT. The paper presents a method of the systematic development of correct programs. A program is called correct if it is partially correct wrt given pre- and post-conditions and if it neither loops indefinitely nor aborts. The requirement of non-abortion makes our correctness stronger than the so called total correctness which is usually understood as partial correctness plus non-looping. In the described method programs are developed and transformed together with their documentation. The documentation consists of a precondition, a postcondition and a set of assertions. The assertions are choosen in such a way that they may be used in the correctness proof of the program. This provides an adequate description of the algorithm and may also be useful in program testing. The rules of program derivation are sound, i.e. if applied to correct programs they yield correct programs. The application of the method is explained on the example of the derivation of a bubblesort program.

1. INTRODUCTION

The motivation for the structured programming (Dijkstra 1968) was to help the programmer in developing, understanding, documenting and possibly also proving correct the program. The latter goal, however important and worth of effort - see Dijkstra (1976) for an interresting discussion of this problem - is still quite cumbersome at least if we first develop the program and only then try to prove it correct. The idea of developing and proving programs simultaneously stimulated many authors to formalize the process of programming by describig its steps as more or less formal transformations (Dijkstra (1975), Darlington (1975, 76), Spitzen, Levitt and Lawrence (1976), Wegbreit (1976), Bär (1977); Burstal and Darlington (1977) Bjørner (1978)). In such an approach every step of a proof of correctness has the same scheme: we prove the correctness of the current version of the program knowing that all former versions were correct. This observation rises immediately a new idea. Instead of checking each time that our programming step does not violates the correctness, we can prove once and for all that in a certain class of programs this step always preserves the correctnes (Dershowitz and Manna (1975), van Emden (1975), Irlik (1976),Blikle (1977A, B, 78), Back (1978)). In this way the correctness proofs of programs are replaced by the soundness proofs of transformation rules.

In developing programs by sound transformations we successfully avoid the necessity of proving programs correct but at the same time we lose of course, the unique oportunity of learning from the proof of correctness about many relevant properties of the program (cf.Dijkstra (1976)). Even if these properties may be implicitly seen by the programmer through the way he has developed the program, they certainly will not be seen by the user since they are not reflected - or even not reflectable - in the specification of programs by pre- and postconditions.

In order to maintain all the advantages of programming by sound transformations without losing the advantages of having the proof of correctness in an explicite form we propose in this paper to enrich the input-output specification of programs by the specification of the proof of correctness. Technically, the proof of correctness is specified by a set of assertions nested in the appropriate places between the instructions of the program. Program correctness is understood here as partial correctness plus non-looping and non-abortion. Such correctness is stronger than so called total correctness which usually means (cf.Manna and Pnueli (1974) and Manna (1974)) partial correctness plus non-looping. The fact that we deal with the abortion problem makes the clasical logic inadequate for the treatment of conditions and assertions in our method. We are using therefore McCarthy's (1967) partial logic which perfectly fits to that goal.

Another method of developing programs with assertions has been described by Lee, de Roever and Gerhard (1979). In that method, however, the underlined concept of program correctness is the partial correctness and the assertions coincide with Floyd's invariants.

The fact that we extend the I/O specification of programs (the pre- and postconditions) by adding assertions seems to have the following advantages: First of all, assertions describe local properties of programs which may be helpful not only in program understanding but also in its maintainance and testing. Secondly, knowing assertions we can recheck program correctness in a nearly mechanical way. This option may be of interest in all these circumstances, where we need an extra high reliability of programs; i.e. in microprogramming. Finally, since our assertions satisfy the requirements of proofs of termination, they adequately describe the time complexity of all loops.

The paper is organized as follows. Sec.2 contains the description of an abstract programming language which provides the experimentation field for the method. Sec.3 is devoted to the particular logical framework which is needed in order to handle the problem of abortions. Sec.4 introduces the concept of the assertion-correctness of programs. The program development rules and the problem of their soundness is discussed in Sec.5. The last Sec.6 contains an example of the application of the method in the development of a bubblesort program. Another such example may be found in Blikle (1978) where an earlier version of the present method was applied to the development of an efficient program computing the integer square root.

2. THE LANGUAGE OF PROGRAMS' DEVELOPMENT AND SPECIFICATION

It is not the aim of this paper to concentrate on the technical details of a programming language suitable for our method of program derivation. All we want to convey to the reader is the general idea of such a language along with some technical suggestions about the transformation rules and the programming techniques. The language which is described below should be considered as only an experimental version. Since it represents a certain method of programming and since it is the first approximation of what we may expect to have in the future, we shall call it PROMETH-1. (programming method, 1).

PROMETH-1 is a language of programs' development and documentation, rather than a simple language for coding algorithms. Consequently we allow there certain abstract constructions and data types which will be used only in the development and the documentation of programs but which are not intended for implementation. Secondly, our language represents, in fact, a family of languages with a common general syntax and semantics but with different abstract data types. In other words, we have the option of user-definable data types. Each time the user intends to derive a program, he starts from the design of an appropriate data type, thus establishing the primitives of the syntax and the sematics of his-problem-oriented version of PROMETH-1.

In this paper by an <u>abstract data type</u> (cf. Guttag (1977), Liskov and Zilles (1975)) we mean a relational system of the form $DT = (D, f_1, \ldots, f_n, q_1, \ldots, q_m)$ where D is a nonempty many-sorted carier and $f_i \in [D^{a_i} \longrightarrow D]$ and $q_j \in [D^{b_j} \longrightarrow \{true, false\}]$ are partial functions and partial predicates respectively. The partiality of functions and predicates is an essential point in our approach and is strongly connected with the fact that we are dealing with the problem of abortion (see Sec.3). The problem of data-type specification in PROMETH-1 is skiped. For the sake of this paper we simply assume that our data type is always somehow defined - e.g. in the set theory. A very elegant formalism for data-type specification is provided by the initial-algebra approach (see ADJ (1975), Goguen (1978), Erig, Kreowski, Padawitz (1978) and papers referenced there).

Given DT we may establish the primitive syntactical components of PROMETH-1. First, with each f_i and q_j we associate the symbols F_i and Q_j respectively. For simplicity we assume that "=" will denote both, the identity relation in D and the corresponding predicate symbol. We also assume that for each sort in D there is in the set of q_j's the corresponding <u>sort predicate.</u> This is a unary total predicate which gives the value true for arguments of the given sort and gives false for all other arguments. Typical sort predicates are <u>integer</u> n, <u>array</u> a, etc. We also want to have constant predicates <u>true</u> and <u>false</u> defined in an obvious way. Of course, these predicates are total as well.

Having introduced the data-type oriented syntax we establish the infinite set of <u>identifiers</u> (individual variables) IDE and we are ready to define the class EXP of <u>expressions</u> and the class CON of <u>conditions</u>. These classes are mutually recursive, i.e. each of them is defined recursively with respect to the other. Formally we should use here a set of BNF equations but for the sake of clarity we restrict ourselves to a more intuitive definition.

EXP is the least syntactical class with the following properties:

1) IDE $\subseteq$ EXP

2) $F_i(E_1,\dots,E_{a_i}) \in$ EXP for any $i \leq n$ and any $E_1,\dots,E_{a_i} \in$ EXP

3) <u>if</u> c <u>then</u> E_1 <u>else</u> E_2 <u>fi</u> $\in$ EXP for any $c \in$ CON and any $E_1, E_2 \in$ EXP

CON is the least syntactical class with the following properties:

1) $Q_j(E_1,\ldots,E_{b_j}) \in$ CON for any $j \leq m$ and any $E_1,\ldots,E_{b_j} \in$ EXP

2) $c_1 \rightarrow c_2, c_3 \in$ CON for any $c_1, c_2, c_3 \in$ CON

3) $(\forall x)c \in$ CON and $(\exists x)c \in$ CON for any $x \in$ IDE and $c \in$ CON

<u>Remark.</u> In the applications we identify F_i with f_i and Q_j with q_j and allow the infix notation. Typical elementary expressions are therefore $x+\sqrt{y}$, $(x+y)-z$, <u>max</u>$\{k \mid k<2^n\}$, etc. and typical elementary conditions are of the form $z \leq y$, a <u>is sorted</u>, i $\leq$ <u>length</u> a, etc. □

Having defined IDE, EXP and CON we can define subsequent syntactical classes: ASR - of <u>assertions</u>, TES - of <u>tests</u>, ASG - of assignments, EIN - of <u>elementary instructions</u>, SIN - of <u>simple instructions</u> and INS - of <u>instructions</u>. We use the BNF formalism for this purpose:

ASR ::= <u>as</u> CON <u>sa</u>

TES ::= <u>if</u> CON <u>fi</u>

ASG ::= IDE := EXP

EIN ::= <u>skip</u> | <u>abort</u> | TES | ASG | EIN;EIN

SIN ::= EIN |
      <u>if</u> CON <u>then</u> INS <u>else</u> INS <u>fi</u> |
      <u>while</u> CON <u>do</u> INS <u>as</u> CON <u>sa</u> EXP <u>od</u>

```
INS ::= SIN |
        SIN as CON sa INS
        inv CON; INS vni
```

The class SIN has been introduced for technical reasons in order to have an unambiguous grammar. This is necessary for the definition of semantics. (Sec.4)

So far we have defined rather usual programming concepts, although with somewhat extravagant syntax. The latter is the consequence of the assumption that our instructions (programs) are enriched by assertions. In the semantics of instructions these assertions play the role of comments and are simply skiped in the execution. Their role becomes essential in the class of assertion specified programs (abbreviated a.s. programs). This class is denoted by ASP and is defined by the equation:

```
ASP ::= pre CON; INS post CON
```

In every a.s. program the conditions following pre and post are called the precondition and the postcondition respectively. In contrast to instructions, which describe algorithms and therefore their semantical meanings are I/O functions, the a.s. programs are claims about algorithms and therefore their semantical meanings are truth values. This is formalised in Sec.4.

In order to define the semantics of our language we need to recall a few elementary facts from the calculus of binary relations. Let D_1, D_2 and D_3 be arbitrary nonempty sets. Given two binary relations $R_1 \subseteq D_1 \times D_2$ and $R_2 \subseteq D_2 \times D_3$ we define their

<u>composition</u> by the equation $R_1R_2 = \{(a,b) \mid (\exists c)(aR_1c \,\&\, cR_2b)\}$. This operation is associative, distributive over arbitrary unions and monotone (w.r.t. inclusion $\subseteq$) in both arguments. Instead of $aR_1c \;\&\; cR_2b$ we frequently write, for short, aR_1cR_2b. The operation of composition may be generalized to the cases where one of the arguments is a set. Let $B \subseteq D_1$, $C \subseteq D_2$ and $R \subseteq D_1 \times D_2$. Then

$$BR = \{a \mid (\exists b)(b \in B \,\&\, bRa\}$$
$$RC = \{a \mid (\exists c)(aRc \,\&\, c \in C\}$$

Of course, BR is the image of B in R and RC is the coimage of C in R. These new operations are also monotone and distributive in both arguments and are weakly associative in the following sense:

$$R_1(R_2C) = (R_1R_2)C \quad \text{for} \quad R_1 \subseteq D_1{\times}D_2,\ R_2 \subseteq D_2{\times}D_3,\ C \subseteq D_3$$
$$(BR_1)R_2 = B(R_1R_2) \quad \text{for} \quad B \subseteq D_1,\ R_1 \subseteq D_1{\times}D_2,\ R_2 \subseteq D_2{\times}D_3$$

By Φ we denote the empty set and the empty relation. Both of them (if at all they are different!) are zeros of the composition.

The case of particular interest is that where $D_1 = D_2 = D$, i.e. where we are considering relations $R \subseteq D{\times}D$. In this case the operation of composition has a neutral element which is the identity relation

$$I = \{(a,a) \mid a \in D\}$$

Given an arbitrary $R \subseteq D \times D$ we define $R^0 = I$, $R^{i+1} = RR^i$ for $i \geq 0$ and $R^* = \cup_{i=0}^{\infty} R^i$. The latter is called the <u>iteration</u> or the <u>reflexive and transitive closure</u> of R.

The first semantical object which we define over DT is the set of <u>states</u> $S = [IDE \longrightarrow D]_t$. According to this equation states are total valuations of the set of identifiers, which means that in our model all the identifiers are global. This assumption can easily by relaxed and was adopted here for technical simplicity. It may be partly justified by the fact that we do not need the concept of a local variable in our example of Sec.6.
In this paper semantics is understood as a function (strictly speaking a many-sorted homomorphism) which assigns meanings to all the investigated syntactical entities. This function is denoted by [ ] hence [X] denotes the meaning of X, where X may be an expression, a condition, an instruction etc. Of course, depending on the class where X belongs, [X] will be of appropriate type:

1) $[\ ]: EXP \longrightarrow [S \longrightarrow D]$
2) $[\ ]: CON \longrightarrow [S \longrightarrow \{true, false\}]$
3) $[\ ]: INS \longrightarrow [S \longrightarrow S]$
4) $[\ ]: ASP \longrightarrow \{true, false\}$

Here and in the sequel $[X \longrightarrow Y]$ denotes the set of all partial functions from X to Y.
The semantics of the class EXP is defined by the following recursive (schemes of) equations:

1) $[x](s) = s(x)$
2) $[F_i(E_1,\ldots,E_{a_i})](s) = f_i([E_1](s),\ldots,[E_{a_i}](s))$

3) $[\underline{\text{if}}\ c\ \underline{\text{then}}\ E_1\ \underline{\text{else}}\ E_2\ \underline{\text{fi}}](s) = \begin{cases} [E_1](s) & \text{if } [c](s) = \text{true} \\ [E_2](s) & \text{if } [c](s) = \text{false} \\ \text{undefined} & \text{if } [c](s) \text{ undefined} \end{cases}$

This coincides with the usual understanding of expressions both in programming languages and in mathematical logic. In the semantics of CON we have to comply with a rather unusual assumption that our conditions represent partial functions too. Since this requires an additional discussion we postpone the description of the semantics of CON to Sec.3.

Once we have established the semantics of EXP and CON the semantics of INS is defined by the denotational equations listed below. For the convenience of wording we assume that x, E, c and IN possibly with indices will always denote identifiers, expressions, conditions and instructions respectively.

$$(1) \quad [\underline{\text{as}}\ c\ \underline{\text{sa}}] = I \qquad (2.2)$$

In other words, assertions are semantically equivalent to skip (e.g. as comments in ALGOL 60).

$$(2) \quad [\underline{\text{if}}\ c\ \underline{\text{fi}}] = \{(s,s) \mid [c](s) = \text{true}\}$$

This means that if c fi is a side-effect-free test which results a skip if c is satisfied and which aborts the execution whenever either ~c is satisfied or the value of c is undefined.

$$(3) \quad [x:=E] = \{(s_1,s_2) \mid s_2(x) = [E](s_1) \text{ and } s_2(y) = s_1(y) \text{ for all } y \in IDE-\{x\}\}$$

(4) [skip] = I

(5) [abort]= Φ

(6) $[IN_1;IN_2] = [IN_1][IN_2]$

(7) [if c then IN_1 else IN_2 fi] = [if c fi][IN_1] $\cup$ [if ~c fi][IN_2]

(8) [while c do IN as c_a sa E od] = ([if c fi][IN])*[if ~c fi]

(9) [IN_1 as c sa IN_2] = [IN_1][IN_2]

(10) [inv c; IN vni] = [IN]

The semantics of the class ASP of assertion specifies programs strongly relates to the semantics of conditions and therefore is postponed to Sec.4.

3. ON THE PARTIALITY OF CONDITIONS AND THE UNDERLINED LOGIC

In the majority of approaches to the problem of program correctness one may find the assumption that the expressions and conditions represent total functions. This assumption considerably simplifies the mathematical model but from the practical point of view is hardly acceptable. Every programmer knows that both expressions and conditions may lead to abortion if evaluated in an improper environment. For instance we frequently cannot evaluate division on integers and we certainly cannot evaluate the condition $a(i) \leq a(j)$ whenever either i or j is outside of the scope of a. Whereas the first case can easily be detected on the syntactical level (in compile time), the second requires the semantical analysis.

The partiality of expressions is something to which we already got used in mathematics, e.g. in the theory of recursive functions, and therefore it does not require any particular explanation. The partiality of conditions, however, has not been so widely accepted although the need of it in the theory of programs was recognized as early as in 1961 by J.McCarthy (see McCarthy 1967). We take the McCarthy's model as the base for our definition of the semantics of CON.

Similarly as for the case of EXP (Sec.2) the semantics of CON is defined by a set of (schemes of) recursive equations

1) $[Q_j(E_1,\dots,E_{b_j})](s) = q_j([E_1](s),\dots,[E_{b_j}](s))$.

2) $[c_1 \rightarrow c_2,c_3](s) = \begin{cases} [c_2](s) & \text{if } [c_1](s) = \text{true} \\ [c_3](s) & \text{if } [c_1](s) = \text{false} \\ \text{undefined} & \text{if } [c_1](s) \text{ undefined} \end{cases}$

3) $[(\forall x)c](s) =$
- true if for any state s_1 which differs from s at most in x, $[c](s)$ = true
- false if there exists a state s_1 which differs from s at most in x, such that $[c](s_1)$ = false
- undefined in all other cases, i.e. if there is no state s_1 which differs from s at most in x such that $[c](s)$=false but for some states s_1 which differ from s at most in x, $[c](s)$ is undefined.

$$4)\ [(\exists x)c](s) = \begin{cases} \text{true} & \text{if there exists a state } s_1 \text{ which differs from } s \text{ at most in } x, \text{ such that } [c](s) = \text{true} \\ \text{false} & \text{if for any state } s_1 \text{ which differs from } s \text{ at most in } x,\ [c](s_1) = \text{false} \\ \text{undefined in all other cases} & \end{cases}$$

These equations require a few comments. First of all observe that the evaluation of the condition $c_1 \to c_2, c_3$ is similar to the evaluation of the if-then-else expressions. If c_1 is undefined, then the whole condition is undefined. If c_1 is true, then we evaluate c_2 regardless whether c_3 is defined or not. The same concerns the symmetrical case. For instance $x \geq 0 \to x+1>0,\ x^{-1}<0$ is true for any state s such that $s(x) = 0$ despite the fact that $[x^{-1}<0](s)$ is undefined. Some important consequences of this property of $\to$ will be discussed later in this section. Now let us concentrate on a few examples with quantifiers. Suppose that the carier D of our data type contains two sorts - integers and integer arrays, and consider a state s such that for a certain identifier y, [integer y](s) = true.

Then

1) $[(\forall x)(x+y)^2>0](s) = \text{false}$
2) $[(\forall x)(x+y)^2 \geq 0](s)$ is undefined
3) $[(\forall x)(\underline{\text{integer}}\ x \to (x+y)^2 \geq 0, \underline{\text{true}})](s) = \text{true}$

4) $[(\exists x)(x+y)^2 \leq 0](s) = \text{true}$
5) $[(\exists x)(x+y)^2 < 0](s)$ is undefined
6) $[(\exists x)(\underline{\text{integer}}\ x \to (x+y)^2 < 0, \underline{\text{false}})](s) = \text{false}$

In the examples 3) and 6) the quantifiers are restricted to a certain sort. Since this is a very common case, it is worth to extend the syntax of CON by allowing the conditions of the form $(\forall\ \underline{sort}\ x)c$ and $(\exists\ \underline{sort}\ x)c$, with the following semantics:

$$[(\forall\ \underline{sort}\ x)c] = [(\forall x)(\underline{sort}\ x \rightarrow c, \underline{true})]$$
$$[(\exists\ \underline{sort}\ x)c] = [(\exists x)(\underline{sort}\ x \rightarrow c, \underline{false})]$$

Now, 3) and 6) can be written in a more readable way:

7) $[(\forall\ \underline{integer}\ x)(x+y)^2 \geq 0](s) = \text{true}$

8) $[(\exists\ \underline{integer}\ x)(x+y)^2 < 0](s) = \text{false}$

For further convenience we may extend CON again by allowing the usual connectives such as $\vee$, &, $\sim$ and $\supset$. We define their semantics after McCarthy (1967):

1) $[c_1 \vee c_2] = [c_1 \rightarrow \underline{true}, c_2]$

2) $[c_1\ \&\ c_2] = [c_1 \rightarrow c_2, \underline{false}]$ (3.1)

3) $[\sim c_1] = [c_1 \rightarrow \underline{false}, \underline{true}]$

4) $[c_1 \supset c_2] = [c_1 \rightarrow c_2, \underline{true}]$

These connectives constitute a natural generalization of the classical case. Indeed, if the values of both c_1 and c_2 are defined, then the values of 1) - 4) are the same as in the classical logic. If c_1 is undefined, then each of 1)-4) is undefined but if c_1 is defined, then 1),2) and 4) may be defined even if c_2 is undefined. This asymetry may be interpreted as the consequence of the fact that in our semantics we execute the conditions from left to right. E.g. if we execute $c_1 \vee c_2$ and the value of

c_1 turns out to be true, then we do not care about c_2. Due to this principle neither $\vee$ nor $\&$ is commutative in McCarthy's logic.

In our approach to programming we frequently have to describe certain relations which may hold between conditions. For this sake we first introduce an auxiliary notation. Let for any c

$$\{c\} = \{s \mid [c](s) = \text{true}\}$$

As is easy to prove, for any c_1 and c_2

$$\{c_1 \& c_2\} = \{c_1\} \cap \{c_2\}$$
$$\{c_1 \vee c_2\} \subseteq \{c_1\} \cup \{c_2\}$$

Now, we define four relations in the set CON:

$c_1 \simeq c_2$ if $[c_1] = [c_2]$ read: c_1 is <u>strongly equivalent</u> to c_2

$c_1 \sqsubseteq c_2$ if $[c_1] \subseteq [c_2]$ read: c_1 is <u>less defined than</u> c_2

$c_1 \Longleftrightarrow c_2$ if $\{c_1\} = \{c_2\}$ read: c_1 <u>is equivalent</u> to c_2

$c_1 \Longrightarrow c_2$ if $\{c_1\} \subseteq \{c_2\}$ read: c_1 <u>implies</u> c_2

Our strong equivalence coincides with the McCarthy's strong equivalence but our equivalence is not his weak equivalence.
The set CON may be regarded as a relational system with the operations $\simeq$, $\sqsubseteq$, $\Longleftrightarrow$, $\Longrightarrow$. Below we sketch some properties of this system which we shall need in the applications. Proofs are left to the reader. Here and in the sequel we adopt the convention of using the words <u>equivalent</u> and <u>implies</u> homonymously: in the sense atta-

ched to $\iff$ and $\implies$ and in a colloquial sense, e.g. in saying that $c_1 \sqsubseteq c_2$ implies $c_1 \implies c_2$. The appropriate meaning will be always defined by the context.

THEOREM 3.1 The relations $\eqsim$ and $\iff$ are equivalence relations in CON. Moreover $\eqsim$ is a congruence, but $\iff$ is not. □

The relation $\iff$ is not a congruence since $c_1 \iff c_2$ does not imply $\sim c_1 \iff \sim c_2$

THEOREM 3.2 The relations $\sqsubseteq$ and $\implies$ are partial orderings in $CON/_{\eqsim}$ and $CON/_{\iff}$ respectively. The operations $\vee$ and $\&$ are monotone wrt both these orderings and the remaining operations are monotone only wrt $\sqsubseteq$. □

THEOREM 3.3 The equivalence $\eqsim$ is strictly stronger than $\iff$, i.e. $c_1 \eqsim c_2$ implies $c_1 \iff c_2$ but not vice versa. Also the ordering $\sqsubseteq$ is strictly stronger than $\implies$, i.e. $c_1 \sqsubseteq c_2$ implies $c_1 \implies c_2$ but not vice versa. □

Below we are listing some important equivalences and inequalities of the propositional calculus in CON:

(1a) $(c_1 \vee c_2) \vee c_3 \eqsim c_1 \vee (c_2 \vee c_3)$

(1b) $(c_1 \& c_2) \& c_3 \eqsim c_1 \& (c_2 \& c_3)$

(2a) $c_1 \vee c_1 \eqsim c_1$

(2b) $c_1 \& c_1 \eqsim c_1$

(3a) $c_1 \vee (c_1 \& c_2) \eqsim c_1$

(3b) $c_1 \& (c_1 \vee c_2) \eqsim c_1$

(4a) $c_1 \& (c_2 \vee c_3) \eqsim (c_1 \& c_2) \vee (c_1 \& c_3)$

(4b) $c_1 \vee (c_2 \,\&\, c_3) \eqsim (c_1 \vee c_2) \,\&\, (c_1 \vee c_3)$

(5a) $c_1 \vee \underline{\text{false}} \eqsim c_1$

(5b) $c_1 \,\&\, \underline{\text{true}} \eqsim c_1$

(6) $\sim(\sim c_1) \eqsim c_1$

(7a) $\sim(c_1 \vee c_2) \eqsim \sim c_1 \,\&\, \sim c_2$

(7b) $\sim(c_1 \,\&\, c_2) \eqsim \sim c_1 \vee \sim c_2$

(8a) $c_1 \vee \sim c_1 \sqsubseteq \underline{\text{true}}$

(8b) $c_1 \,\&\, \sim c_1 \sqsubseteq \underline{\text{false}}$

(9a) $\sim(\exists x)c \eqsim (\forall x)(\sim c)$

(9b) $\sim(\forall x)c \eqsim (\exists x)(\sim c)$

This proves that McCarthy's calculus with the strong equivalence is quite similar to the clasical propositional calculus. So far we have discovered just two exceptions: (1) the lack of the commutativity of $\vee$ and $\&$, and (2) the inequalities in the place of equivalences in (8a) and (8b). On the strength of Theorem 3.3 we can replace $\eqsim$ by $\Longleftrightarrow$ in (1a)-(7b) and $\sqsubseteq$ by $\Longrightarrow$ in (8a), (8b). There are also some laws which hold for $\Longleftrightarrow$ and $\Longrightarrow$ but does not hold for $\eqsim$ and $\sqsubseteq$:

(10a) $c_1 \Longrightarrow c_1 \vee c_2$

(10b) $c_1 \,\&\, c_2 \Longrightarrow c_1$

(11) $c_1 \,\&\, c_2 \Longleftrightarrow c_2 \,\&\, c_1$

Here the symmetry between $\vee$ and $\&$ is no more the case. In $\mathrm{CON}/_{\Longleftrightarrow}$, $\&$ is commutative but $\vee$ is not. In particular $c_1 \Longrightarrow c_2 \vee c_1$ does not hold!

The discussion of McCarthy's logical calculus given in this section is far from being complete. We only gave a general outline of the approach restricted to our needs connected with the development of the example of Sec.6. This subject definitely deserves an independent investigation.

4. THE CORRECTNESS AND THE ASSERTION CORRECTNESS OF PROGRAMS

As was already mentioned in Sec.2 the semantical meanings of a.s. programs are truth values. Accordingly to the traditional wording of the field we shall say, however, that an a.s. program is correct rather than true. Below we define two concepts of correctness. The first is the strengthenning of Manna-Pnneli's total correctness and may be understood as describing an auxiliary semntics. The other, called assertion correctness, is the principal concept of correctnes in our method.

An assertion specified program pre c_{pr}; IN post c_{po} is called correct if

$$\{c_{pr}\} \subseteq [IN]\{c_{po}\} \qquad (4.1)$$

This correctness means that for any state s which satisfies c_{pr} the execution of IN terminates successfully - i.e. neither aborts nor runs indefinitely - and the output state satisfies c_{po}. Observe that in the usual understanding of total correctness (Manna and Pnueli (1974), Manna (1974)) the problem of abortion is neglected: successfull termination simply means no indefinite execution. Consequently, the correctness defined by (4.1) is stronger than the total correctness. For better explanation consider the program

$\underline{\text{pre}}$ $\underline{\text{integer}}$ $\underline{\text{array}}$ $A[0{:}n]$ & $a = A$ & $i = n$

$\underline{\text{while}}$ $a(i)<a(i-1)$ $\underline{\text{do}}$ $a := \underline{\text{swap}}\ (a,i,i-1);$

$i := i-1$ $\underline{\text{od}}$

$\underline{\text{post}}$ a $\underline{\text{is a permutation of}}$ A

where swap (a,i,i-1) denotes the result of swapping the i-th element with the i-1 element in a. This program is totally correct (i.e. may be proved correct in the Manna-Pnueli's system) but it is not correct in our sense since i may reach the value of 0 in which case the execution aborts. For further discussion of (4.1) and the corresponding proof techniques see Blikle (1977C,79).

The above defined concept of correctness is restricted to global properties of programs. Below we define the assertion correctness which refers not only to the pre- and postcondition but also to the assertions of the program. Intuitively pre c_{pr}; IN post c_{po} is assertion correct if it is correct and if the assertions which occur in IN may be used in the proof of (4.1). The formal definition is inductive w.r.t. the syntax of INS:

(A) For any elementary instruction IN the a.s. program pre c_{pr}; IN post c_{po} is assertion correct if it is correct. Notice that elementary instructions contain no assertions.

(B) The a.s. program

$$\underline{\text{pre}}\ c_{pr};\ \underline{\text{if}}\ c\ \underline{\text{then}}\ IN_1\ \underline{\text{else}}\ IN_2\ \underline{\text{fi}}\ \underline{\text{post}}\ c_{po}$$

is assertion correct if

(B1) $c_{pr} \Longrightarrow c \vee \sim c$

(B2) pre c_{pr} & c; IN_1 post c_{po} is assertion correct

(B3) pre c_{pr} & ~c; IN_2 post c_{po} is assertion correct

(C) The a.s. program

pre c_{pr}; while c do IN as c_a sa E od post c_{po}

is assertion correct if

(C1) $c_{pr} \Longrightarrow c_a$ & $E \geq 0$

(C2) pre c_a & $E \geq 1$; if c fi; IN post c_a & $E \geq 0$ is a.c.

(C3) pre c_a & $E < 1$; if ~c fi post c_{po} is a.c.

(C4) [if c_a & $E \geq 1$ fi][IN][E] $\subseteq$ [E-1]

This definition requires a few comments. First of all E is here the loop counter i.e. a real expression whose integer value gives the number of cycles through IN which must be performed in order to exit from the loop. This concept may be easily generalized using well founded sets (Floyd (1967)). We do not need, however, this generalization in our example of Sec.6 and moreover the arithmetical loop counter has the advantage of giving the explicite estimation of the time complexity of the loop (see the example in Sec.6). The condition c_a is called the loop assertion and loosely speaking describes the global effect of IN. Under this interpretation (C1) says that for any state which satisfies the precondition c_{pr}, the loop assertion is satisfied and the number of cycles to be performed is defined. (C2) says that whenever the loop assertion is satisfied and the number of remaining cycles is not less than 1, then the body of the loop is executable, the successive state satisfies c_a again and the number of remaining cycles through the loop is defined. It also says that the above property may be proved using the assertions of IN. The conjunction of (C1) with (C2) guarantees that under the precondition c_{pr} the loop will be executed without abortion and c_a will be preserved in each cycle. Two remaining conditions imply that this execution will not continue indefinitely. Indeed, (C3) claims that if c_a is satisfied and the remaining number of cycles is 0 then the control exits the loop and the postcondition is satisfied. The last condition (C4) guarantees that the value of E will fall under 1 in a finite time since any execution of IN in the environment where $c_a \,\&\, E \geq 1$ is satisfied decrements the value of E by 1.

(D) If $IN_1 \in SIN$ then the a.s. program

$$\underline{\text{pre}}\ c_{pr};\ IN_1\ \underline{\text{as}}\ c_a\ \underline{\text{sa}}\ IN_2\ \underline{\text{post}}\ c_{po}$$

is <u>assertion correct</u> if

(D1) $\underline{\text{pre}}\ c_{pr};\ IN_1\ \underline{\text{post}}\ c_a$ is assertion correct

(D2) $\underline{\text{pre}}\ c_a;\ IN_2\ \underline{\text{post}}\ c_{po}$ is assertion correct

(E) The a.s. program

$$\underline{\text{pre}}\ c_{pr};\ \underline{\text{inv}}\ c_i;\ IN\ \underline{\text{vni}}\ \underline{\text{post}}\ c_{po} \qquad (4.2)$$

is <u>assertion correct</u> if the a.s. program $\underline{\text{pre}}\ c_{pr};\ IN_1\ \underline{\text{post}}\ c_{po}$

where IN_1 results in from IN by the substitution for each assertion <u>as</u> c_a <u>sa</u> in IN the assertion <u>as</u> c_a & c_i <u>sa</u>, is assertion correct.

The condition c_i in (E) is called the <u>permanent invariant</u> in (4.2) and <u>inv</u> c_i is called its <u>declaration</u>. The mirror key-word <u>vni</u> defines the <u>scope</u> of this declaration. Permanent invariants are used to "factorize" conditions which are permanently satisfied in a segment of a program. Typical factorizable conditions are these which describe the unchangable properties of the environment, e.g. the type of identifiers. More examples are provided in Sec.6.

To complete the definition of assertion correctness observe that every assertion in an assertion correct program is a Floyd's invariant but not vice versa. The critical point is that the Floyd invariants usually do not guarantee the executability and the termination of IN. Indeed, consider the a.s. program

<u>pre</u> <u>real</u> a & $a \geq 0$ & $x=a$

    $x:=x+1$

    <u>as</u> $x=a+1$ & $x>0$ <u>sa</u>

    $x:=x^{-1}$

<u>post</u> $x=(a+1)^{-1}$

which is, of course, assertion correct. In the proof of partial correctness of this program we could use the invariant $x=a+1$. This invariant is, however, too weak to prove nonabortion and therefore it is not an assertion in our sense.

One of our motivations in defining the concept of assertion correct program was to formalize the property that a given set of assertions can be used in proving a given program correct. To make sure that our goal has not been missed we must prove, first of all, that every assertion-correct program is correct. The proof of this theorem will also indicate in which way our assertions may be used in the proofs of program correctness.

THEOREM 4.1 Every a.s. program which is assertion correct is correct. □

PROOF. This must be proved by induction on the syntactical complexity of a.s. programs. The first step (case (A) of the definition) is obvious. In the induction step we must consider the cases (B)-(E). Since the only nontrivial case is (C) consider the a.s. program

$$\underline{\text{pre}}\ c_{pr};\ \underline{\text{while}}\ c\ \underline{\text{do}}\ \text{IN}\ \underline{\text{as}}\ c_a\ \underline{\text{sa}}\ E\ \underline{\text{od}}\ \underline{\text{post}}\ c_{po} \qquad (4.3)$$

and assume that it is assertion correct. Now, let for any integer $i \geq 0$, $A_i = \{c_a \,\&\, i \leq E < i+1\}$. We shall show the following

1) $\{c_{pr}\} \subseteq \bigcup_{i=0}^{\infty} A_i$
2) $(\forall i \geq 1)(A_i \subseteq [\underline{\text{if}}\ c\ \underline{\text{fi}};\ \text{IN}]A_{i-1})$
3) $A_o \subseteq [\underline{\text{if}}\ {\sim}c\ \underline{\text{fi}}]\{c_{po}\}$.

The conditions 1) and 3) are immediate from (C1) and (C3) respectively. Prove 2). Let $i \geq 1$ and let $s \in A_i$. Then $s \in \{c_a\}$ and $[E](s) = d$ for some d with $i \leq d < i+1$. By (C2) and the induction assumption we have

$$s \in [\underline{if}\ c\ \ \underline{fi}][IN]\{c_a \,\&\, E \geq 0\}$$

hence there exists s_1 with $s[IN]s_1$, $s_1 \in \{c_a\}$ and $[E](s_1) = d_1$ for some $d_1 \geq 0$. Since, of course

$$s[\underline{if}\ c\ \ \underline{fi}]s[IN]s_1[E]d_1$$

we get by (C4), $[E-1](s) = d_1$. Therefore, $[E](s_1) = d_1 = [E-1](s) = [E](s)-1 = d-1$. This implies the inequalities $i-1 \leq [E](s_1) < i$ which implies $s_i \in A_i$ and terminates the proof of 2). Now, from 2) and 3) we prove by induction on $i \geq 0$

$$A_i \subseteq ([\underline{if}\ c\ \underline{fi};IN])^i[\underline{if}\ {\sim}c\ \underline{fi}]\{c_{po}\}.$$

Therefore, by 1), $\{c_{pr}\} \subseteq \cup_{i=0}^{\infty}([\underline{if}\ c\ \underline{fi};IN])^i[if\ {\sim}c\ \underline{fi}]\{c_{po}\} = ([\underline{if}\ c\ \underline{fi}][IN])^*[\underline{if}\ {\sim}c\ \underline{fi}]\{c_{pr}\}$ which completes the proof by the semantical axiom (8) of Sec.2. □

As was mentioned in Sec.1 our assertions may be useful not only in the documentation and the mathematical verification of programs, but also in program testing. The latter follows from the fact that each assertion describes the local properties of programs, hence an a.s. program may be tested not only against a pre- and post conditions but also against the local assertions. Technically this may be done by the execution of a modified program which results in from the original one by the replacement of every assertion <u>as</u> c_a <u>sa</u> (case D) by the test <u>if</u> c_a <u>fi</u> and every loop assertion with the expression <u>as</u> c_a <u>sa</u> E (case C) by the test <u>if</u> $c_a \,\&\, E \geq 0$ <u>fi</u>. If we call such a program a <u>testing copy</u> of the original program then the following obvious theorem may be proved.

THEOREM 4.2. If an a.s. program is assertion correct, then the corresponding testing copy is correct. □

The obvious proof is left to the reader. Of course, we tacitly assume that the syntax of the language has been extended in such a way that the testing copies of programs belong to the language too. This requires also an obvious extension of semantics.

5. THE RULES OF THE COMPOSITION AND THE TRANSFORMATION OF PROGRAMS.

The main motivation for our method was to provide sound rules of programming. In this section we show a few such rules which seem to have a fairly broad field of applications. By no means, however, should our set of rules be regarded as complete. To get started we give three technical lemmas. Proofs are left to the reader.

LEMMA 5.1. For any identifier k and instruction IN the properties

(i) $[IN][k] \subseteq [k]$ and

(ii) $(\forall s_1,s_2)(s_1[IN]s_2 \Longrightarrow [k](s_1) = [k](s_2))$

are equivalent. □

This lemma says that the property $[IN][k] \subseteq [k]$ may be read as <u>IN does not change the value of k</u>. E.g. $[x:=x+y][y] \subseteq [y]$.

Of course, we can easily generalize this lemma to the case where k stands for an arbitrary expression.

LEMMA 5.2 For any function $F \in [S \longrightarrow S]$ and any $B_1, B_2, C_1, C_2 \subseteq S$ if

$$B_1 \subseteq FC_1 \quad \text{and}$$
$$B_2 \subseteq FC_2$$

then $B_1 \cap B_2 \subseteq F(C_1 \cap C_2)$. □

LEMMA 5.3 For any function $F \in [S \longrightarrow S]$ and any $C_1, C_2 \subseteq S$,

$$FC_1 \cap FC_2 \subseteq F(C_1 \cap C_2) \qquad □$$

Returning to the sound rules of programming we may first of all observe that the definition of assertion correctnes (A)-(E) in Sec.4 already provides five such rules. The rule which follows from (D) is a bit too restricted since it requires that the first component of the composition be a simple instruction. This restriction was introduced only for the sake of the unambiguity of the definition (D) and may be relaxed now:

THEOREM 5.1 For any two instructions IN, and IN_2, if

$$\underline{\text{pre}}\ c_1;\ IN_1\ \underline{\text{post}}\ c_2$$
$$\underline{\text{pre}}\ c_2;\ IN_2\ \underline{\text{post}}\ c_3$$

are assertion correct, then

$$\underline{\text{pre}}\ c_1;\ IN_1\ \underline{\text{as}}\ c_2\ \underline{\text{sa}}\ IN_2\ \underline{\text{post}}\ c_3$$

is assertion correct. □

PROOF. If IN_1 is simple then the proof is done by the definition. Let then IN_1 be arbitrary. In this case IN_1 must be of the form

$$IN^1\ \underline{\text{as}}\ c^1\ \underline{\text{sa}}\ IN^2\ \underline{\text{as}}\ c^2\ \underline{\text{sa}}\ \ldots\ IN^k\ \underline{\text{as}}\ c^k\ \underline{\text{sa}}\ IN^{k+1}$$

for some $k \geq 1$, where all IN^i are simple. This implies that the following programs as assertion correct

$$\underline{\text{pre}}\ c_1;\ IN^1\ \underline{\text{post}}\ c^1$$
$$\underline{\text{pre}}\ c^1;\ IN^2\ \underline{\text{post}}\ c^2$$
$$\ldots$$
$$\underline{\text{pre}}\ c^k;\ IN^{k+1}\ \underline{\text{post}}\ c_2$$

Combining these a.s. programs according to the rule (D) we successively get the following assertion correct programs:

$$\underline{\text{pre}}\ c^k;\ IN^{k+1}\ \underline{\text{as}}\ c_2\ \underline{\text{sa}}\ IN_2\ \underline{\text{post}}\ c_3$$
$$\underline{\text{pre}}\ c^{k-1};\ IN^k\ \underline{\text{as}}\ c^k\ \underline{\text{sa}}\ IN^{k+1}\ \underline{\text{as}}\ c_2\ \underline{\text{sa}}\ IN_2\ \underline{\text{post}}\ c_3$$

etc. □

Besides the techniques of program development resulting from the rules (A)-(E) of Sec.4 there is another important class of techniques which we shall refer to as the _introduction of an invariant_. Generally speaking given an assertion correct program $\underline{\text{pre}}\ c_{pr};\ IN\ \underline{\text{post}}\ c_{po}$ and a condition c we say that we are introducing the invariant c into our program if we transform

IN into an IN_1 such that <u>pre</u> c_{pr} & c; IN_1 <u>post</u> c_{po} & c is assertion correct. Below we describe two particular rules of the introduction of an invariant into a <u>while</u> <u>do</u> loop.

THEOREM 5.2 (<u>the postfix enrichment of while do</u>) If

<u>pre</u> c_{pr}; <u>while</u> c <u>do</u> IN <u>as</u> c_a <u>sa</u> E <u>od</u> <u>post</u> c_{po} (5.1)

is assertion correct, then for any $c_1, c_1' \in CON$ and any $IN_1 \in INS$ if

1) <u>pre</u> c_a & c_1 & $E \geq 1$; <u>if</u> c <u>fi</u>; IN <u>post</u> c_a & c_1' is assertion correct
2) <u>pre</u> c_a & c_1'; IN_1 <u>post</u> c_a & $E \geq 0$ & c_1 is assertion correct
3) [<u>if</u> c_a & c_1' & $E \geq 0$ <u>fi</u>][IN_1][E] $\subseteq$ [E]

then

<u>pre</u> c_{pr} & c_1; <u>while</u> c <u>do</u> IN <u>as</u> c_a & c_1' <u>sa</u> IN_1 <u>as</u> c_a & c_1 <u>sa</u> E <u>od</u> <u>post</u> c_{po} & c_1

is assertion correct. □

COMMENT. Since IN violates the required invariant c_1 (assumption 1)), we have to supply the loop body with a recovery instruction IN_1 leading back to c_1 (assumption 2)). To make it sure that the alteration of the loop does not violate the termination property, we assume that IN_1 preserves the value of the loop counter E (assumption 3)). □

PROOF. We have to check that the resulting program satisfies the definition (C) of Sec.4. First observe that (C1) and (C3) follow immediately from the assertion correctness of (5.1). Next, (C2) follows from 1), 2) and Theorem 5.1. It remains (C4) to be proved. By the assertion-correctness of (5.1)

$$[\underline{\text{if}}\ c_a \,\&\, E\geq 1\ \underline{\text{fi}}][IN][E] \subseteq [E-1]$$

Therefore by 3) and the monotonicity of composition we get

$$[\underline{\text{if}}\ c_a \,\&\, c_1 \,\&\, E\geq 1\ \underline{\text{fi}}][IN][\underline{\text{if}}\ c_a \,\&\, c_1' \,\&\, E\geq 0\ \underline{\text{fi}}][IN_1][E] \subseteq [E-1] \quad (5.2)$$

Now, by 1) (C2) and lemma 5.2 we have

$$\{c_a \,\&\, c_1 \,\&\, E\geq 1\} \subseteq [\underline{\text{if}}\ c\ \underline{\text{fi}}][IN]\{c_a \,\&\, c_1' \,\&\, E\geq 0\} \subseteq$$
$$\subseteq [IN]\{c_a \,\&\, c_1' \,\&\, E\geq 0\}$$

This implies

$$[\underline{\text{if}}\ c_a \,\&\, c_1 \,\&\, E\geq 1\ \underline{\text{fi}}][IN][\underline{\text{if}}\ c_a \,\&\, c_1' \,\&\, E\geq 0\ \underline{\text{fi}}] =$$
$$= [\underline{\text{if}}\ c_a \,\&\, c_1 \,\&\, E\geq 1\ \underline{\text{fi}}][IN]$$

By (5.2) we get therefore

$$[\underline{\text{if}}\ c_a \,\&\, c_1 \,\&\, E\geq 1\ \underline{\text{fi}}][IN][IN_1][E] \subseteq [E-1] \qquad \Box$$

THEOREM 5.3 (<u>the prefix enrichment of while-do</u>). If

$$\underline{\text{pre}}\ c_{pr};\ \underline{\text{while}}\ c\ \underline{\text{do}}\ IN\ \underline{\text{as}}\ c_a\ \underline{\text{sa}}\ E\ \underline{\text{od}}\ \underline{\text{post}}\ c_{po}$$

is assertion correct and

1) **pre** c_a & c_1 & $E \geq 1$; **if** c **fi** IN_1 **post** c_a & c_1' is assertion correct

2) **pre** c_a & c_1'; IN **post** c_a & c_1 & $E \geq 0$ is assertion correct

3) $[$**if** c_a & c_1 & $E \geq 1$ **fi**$][IN_1][IN][E] \subseteq [E-1]$

then

pre c_{pr} & c_1; while c **do** IN_1 **as** c_a & c_1' **sa** IN **as** c_a & c_1 **sa** E **od** **post** c_{po} & c_1

is assertion correct. □

COMMENT. This transformation is dual to the former. The new instruction IN_1, which is executed before IN, violates c_1 and the old instruction IN provides the recovery. Since the nonmodification of E by IN_1 does not imply termination in this case, we have to assume that IN_1;IN has the property required in the definition. □

PROOF. The case (C1), (C2) and (C3) as in Theorem 5.2. The case (C4) is obvious by 3). □

The sound rules of programming described so far are either the rules of composition (B)-(D) or are transformations which change the structure of the program. Another large group of rules consists of transformations which only modify conditions in the program, but which do not change the control structure. Below we give three examples of such rules which are commonly used in program derivation.

THEOREM 5.4 If the a.s. program

$$\underline{\text{pre}}\ c_{pr};\ \text{IN}\ \underline{\text{post}}\ c_{po}$$

is assertion correct and $c'_{pr} \Longrightarrow c_{pr}$ and $c_{po} \Longrightarrow c'_{po}$, then

$$\underline{\text{pre}}\ c'_{pr};\ \text{IN}\ \underline{\text{post}}\ c'_{po}$$

is assertion correct. □

The proof is obvious.

THEOREM 5.5. If in an arbitrarry assertion correct program we replace:

1) any <u>while</u>-<u>do</u> or <u>if</u>-<u>then</u>-<u>else</u> condition c by c_1 such that $c \simeq c_1$,

2) any precondition, postcondition or assertion c by c_1 such that $c \Longleftrightarrow c_1$,

then the resulting program is assertion correct. □

The proof follows immediately from the fact that in the semantics of assertion specified programs each branching condition is represented by the truth function [c], whereas each precondition, postcondition or assertion is represented by the set of states {c}. The essential point in this theorem is, however, that we cannot replace $\simeq$ by $\Longleftrightarrow$ in 1). An appropriate example is given in Sec.6. We shall also see in that section that many conditions, appearing in a.s. programs are of the form $c_1 \,\&\, \ldots \,\&\, c_n$, where c_i are elementary. Since & commutes in $CON/_{\Longleftrightarrow}$ but does not commute in $CON/_{\simeq}$ (Sec.3) our theorem indicates that the ordering of c_i s in $c_1 \& \ldots \& c_n$ is irrelevant whenever the latter appears as a precondition, a postcondition or an assertion but becomes relevant if it appears in while-do or if-then-else.

THEOREM 5.6. If the a.s. program

pre c_{pr}; while c do IN as c_a sa E od post c_{po}

is assertion correct, then the a.s. program

pre c_{pr}; while c do IN as c_a sa E od post $c_{po} \& c_a$

is assertion correct. □

The proof is immediate from the definition (C) of Sec.4.

6. 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 constitute an environment which satisfies the following Archimedes' principle: each bubble which is lighter 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 swaping. 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 violates the Archimedes' principle. It is intuitively quite clear that in the last step of the procedure our column of bubbles will be ordered according to the increased weights.

The systematic development of the bubblesort program requires, first of all, the establishment of an appropriate data-type. It will be developed in a stepwise manner along with the development of the program. Since in this paper we skip the problem of the formal specification of data type, we are using below a mixture of formal and intuitive mathematics. In many cases we simply refer to a known mathematical concept (e.g. that of a permutation) rather than give an axiomatic definition. Despite this informality of our approach it still seems advisable to keep the many--sorted algebra style (ADJ 1975) in the specification of sorts and arities of functions. We start by the first approximation of our data-type and program.

SORTS

Int - integers

Arr - arrays; each array is a total function

$a: \{0,\dots,n\} \longrightarrow Int$, where $n \geq 0$

Bol - {true, false}

FUNCTIONS

+ , - , 0, 1 - the arithmetical functions and constants length: Arr ⟶ Int - the length of an array

component: Arr × Int ⟶ Int - the i-th component of an array; according to the common style we shall write a(i) in the place of component(a,i)

seg: Arr × Int ⟶ Arr - the initial segment;

$seg\ (a,j) = (a(0),\ldots,a(j))$ for $0 \le j \le length\ a$

PREDICATES

integer, array - the sort predicates (Sec.2)

$\le$, $<$ - The usual arithmetical inequalities

is sorted: Arr ⟶ Bol

a is sorted: $:\approx$ $(\forall$ integer $i)(0 \le i < length\ a \supset$

$\supset a(i) \le a(i+1))$

perm : Arr × Arr ⟶ Bol

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

Now, we may establish the first approximation of our program which we shall informally call the propulsion loop. Here and in the sequel the operational part of the program will be framed in order to distinguish it visually from the specification part.

```
pre array A & a = A & j = 0 & k = length A
inv k = length a & a perm A & 0≤j≤k
 ┌─────────────────────────┐
 │ while j<k do j := j+1   │                              (P1)
 └─────────────────────────┘
    as true sa k-j od
vni
post j=k
```

This program only defines the framework of further approximations and is, obviously, assertion correct. Into this program we shall introduce the invariant <u>seg</u>(a,j) <u>is sorted</u> using the postfix enrichment of the loop (Theorem 5.2). Let

$$c_1 :\approx \underline{\text{seg}}\ (a,j)\ \underline{\text{is sorted}}$$
$$c_1' :\approx \underline{\text{seg}}\ (a,j-1)\ \underline{\text{is sorted}}\ \&\ j \geq 1$$

and let

$$c :\approx k = \underline{\text{length}}\ a\ \&\ a\ \underline{\text{perm}}\ A\ \&\ 0 \leq j \leq k$$

Of course, c is the permanent invariant declared in P_1. Now, accordingly to Theorem 5.2 we have to check that the program

<u>pre</u> $c\ \&\ c_1\ \&\ k-j \geq 1$;

<u>if</u> j<k <u>fi</u>; j:= j+1

<u>post</u> $c\ \&\ c_1'$

is assertion correct and we have to construct an instruction IN_1 such that the following two conditions are satisfied:

$$\underline{\text{pre}}\ c\ \&\ c_1';\ IN_1\ \underline{\text{post}}\ c\ \&\ c_1\ \&\ k-j \geq 0 \quad \text{is a.c.} \qquad (6.1)$$
$$[\underline{\text{if}}\ c\ \&\ c_1'\ \&\ k-j \geq 0\ \underline{\text{fi}}][IN_1][k-j] \subseteq [k-j] \qquad (6.2)$$

The first requirement is, of course, satisfied. Therefore, on the strength of Theorem 5.2, for any IN_1 which satisfies (6.1) and (6.2) the subsequent program is assertion correct. We write it already in a simplified form removing c_1 from the precondition -

since for $j=0$ it is always true - and replacing $j=k$ & c_1 in the postcondition by $j=k$ & a is sorted, since $j=k$ & k = length a & seg(a,j) is sorted implies $j=k$ & a is sorted. Formally we apply here the Theorems 5.4 and 5.6.

```
pre array A & a=A & j=0 & k=length A
inv c
 ┌────────────────────┐
 │ while  j<k  do     │
 │   j:=j+1           │                                (P2)
 └────────────────────┘
   as seg (a,j-1) is sorted & j≥1 sa
 ┌────────────────┐
 │   IN1          │
 └────────────────┘
   as seg (a,j) is sorted sa  k-j  od
vni
post j=k & a is sorted
```

Since there are many IN_1 which satisfy the conditions (6.1) and (6.2), our P_2 represents a class of sorting procedures organized accordingly to the following iterative scheme: given an array a where seg(a,j) has already been sorted, increase j by 1 and permute a in such a way that the new seg(a,j) is sorted again. Our prospective bubblesort belongs to this class. In order to describe it we extend our data type by two new sorts, four new functions and one new predicate

SORTS

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

Set - finite subsets of Int

FUNCTIONS

<u>swap</u>: Arr × Int × Int $\longrightarrow$ Arr;

<u>swap</u> (a,i,j) is, for $0 \leq i,j \leq$ <u>length</u> a, the result of swapping the i-th with the j-th element in a

<u>but</u>: Arr × Int $\longrightarrow$ Vec

a <u>but</u> i is, for $0 \leq i \leq$ <u>length</u> a, the restriction of array a to the domain

{0,...,<u>length</u> a} - {i}

<u>max</u>: Set $\longrightarrow$ Int

<u>max</u> B is the maximal element of the set B

<u>bd</u>: Arr × Int $\longrightarrow$ Int; read: <u>bubbledepth</u>

<u>bd</u>(a,i) = <u>if</u> $i \leq 0 \vee a(i) \geq a(i-1)$ <u>then</u> 0

<u>else</u> <u>max</u> $\{d \mid a(i) < a(i-d)\}$

PREDICATES

First we extend the formerly defined predicate <u>is sorted</u> to the sort of vectors. We also assume that the empty vector satisfies this predicate. Now, we define the new predicate.

<u>bubbles in seg</u>(,): Int×Int×Arr ⟶ Bol

i <u>bubbles in seg</u>(a,j) :≃ $0 \leq i \leq j \leq$ <u>length</u> a &
<u>seg</u>(a,j) <u>but</u> i <u>is sorted</u> &
$i<j \supset a(i+1)>a(i)$

The following may be proved easily:

<u>bd</u>(a,i)≥1 ≈ i>0 & a(i)<a(i-1) (6.3)

i=j & j≥1 & i <u>bubbles in seg</u>(a,j) ⟺
⟺ i=j & j≥1 & <u>seg</u>(a,j-1) <u>is sorted</u> (6.4)

<u>bd</u>(a,i) = 0 & i <u>bubbles in seg</u>(a,j) ⟹
⟹ <u>seg</u>(a,j) <u>is sorted</u> (6.5)

Using the predicate i <u>bubbles in seg</u>(a,j) we may construct the assertion-specified program which describes the bubbling process:

```
pre c & i=j & j≥1 & i bubbles in seg(a,j)
inv c
┌─────────────────────────┐
│ while bd(a,i)≥1 do      │
│    a := swap(a,i-1,i)   │
└─────────────────────────┘
     as i-1 bubbles in seg(a,j) sa                      (P3)
   ┌──────────┐
   │ i := i-1 │
   └──────────┘
     as i bubbles in seg(a,j) sa bd(a,i) od
vni
post c & bd(a,i) = 0 & i bubbles in seg(a,j)
```

The assertion correctness of this program may be proved directly from the definitions (C) and (D) of Sec.4. This proof is left to the reader. Now, we modify (P_3) into the form required by the conditions (6.1) and (6.2). This is done in the following steps.

(1) The pre- and postcondition are modified on the strength of (6.4) and (6.5); cf. Theorem 5.4.

(2) The while condition bd(a,i)≥1, which is inacceptable from the practical viewpoint, is replaced by i>0 & a(i)<a(i-1); cf. (6.3) and Theorem 5.5.

(3) The program which results in from (1) and (2) is combined sequentially (rule (D) of Sec.4) with the program

```
pre c & j≥1 & seg(a,j-1) is sorted
 | i := j |
post c & i=j & j≥1 & seg(a,j-1) is sorted
```

We get

```
pre c & j≥1 & seg(a,j-1) is sorted
inv c
 | i := j |
  as i=j & j≥1 & seg(a,j-1) is sorted sa
 | while i>0 & a(i)<a(i-1) do
 |   a := swap(a,i-1,i)                                   (P4)
    as i-1 bubbles in seg(a,j) sa
   | i := i-1 |
    as i bubbles in seg(a,j) sa bd(a,i) od
vni
post c & seg(a,j) is sorted
```

This program is assertion correct since it has been derived from another assertion correct programs using sound transformations. Since $c \Longrightarrow k-j \geq 0$, the latter condition may be added to the post-condition of (P_4). Therefore, the instruction of (P_4) satisfies (6.1). It also satisfies (6.2) since neither j nor k is modified in (P_4). Consequently, the instruction of (P_4) has all the properties required from IN_1 of (P_2) and may be substituted there. In this way we get the final version of our program:

```
pre array A & a=A & j=0 & k = length A
inv k = length A & a perm A & 0≤j≤k
 ┌──────────────┐
 │while j<k do  │
 │  j := j+1    │
 └──────────────┘
    as j≥1 & seg(a,j-1) is sorted sa
   ┌────────┐
   │ i := j │
   └────────┘
    as i=j & j≥1 & seg(a,j-1) is sorted sa
   ┌──────────────────────────┐
   │while i>0 & a(i)<a(i-1) do│
   │  a := swap(a,i-1,i)      │
   └──────────────────────────┘
      as i-1 bubbles in seg(a,j) sa
     ┌──────────┐
     │i := i-1  │
     └──────────┘
      as i bubbles in seg(a,j) sa bd(a,i) od
    as seg(a,j) is sorted sa k-j od
vni
post j=k & a is sorted
```

This program is, of course, assertion correct. Observe that if in the inner loop we replace the while condition $i>0$ & $a(i)<a(i-1)$ by the condition $a(i)<a(i-1)$ & $i>0$ which is equivalent - but not strongly equivalent - to the former, then we get a program which is no longer correct. That new program aborts whenever the value of i reaches 0 since in that case a(i)<a(i-1) cannot be evaluated.

7. FINAL REMARKS

Practically all the issues discussed in this paper has only been sketched and require further development. First of all we should learn more about McCarthy's logic. Secondly, we should better develop the set of program derivation rules. Thirdly, the language PROMETH-1 should be extended by new programming constructions and by the appropriate subset of data-type specification language. Finally, some attention should be given to the methodology of programming in PROMETH.

ACKNOWLEDGEMENTS

I wish to express my thanks to Hans-Eckart-Sengler, who discovered a mistake in an early version of my example of Sec.6. In order to correct this mistake I decided to introduce a partial logic into PROMETH. It was Antoni Mazurkiewicz, who suggested me to use McCarthy logic in that place. I am deeply greatful to him for this advise. I also wish to express my grattitude to Krzysztof Apt with whome I have discussed some general logical problem of this approach. Finally, my thanks are also addressed to Andrzej Tarlecki, who pointed out another mistake in my example and conveyed to me many interesting remarks about the earlier version of the paper.

REFERENCES

Back, R.J. On the correctness of refinement steps in program development, Dept.of Computer Science, University of Helsinki, Report A-1978-4.

Bär, D.(1977) A methodology for simultaneously developing and verifying PASCAL programs, in: Constructing Quality Software (Proc.IFIP TC-2 Working Conf.,May 1977, Novosybirsk), North Holland, Amsterdam 1978.

Bjørner, D. The Vienna development method (VDM): Software specification & program synthesis. In: Mathematical Studies of Information Processing (Proc.Int.Conf.Kyoto, August 1978), 307-340 to appear in LNCS by Springer Verlag.

Blikle, A.(1977A) A mathematical approach to the derivation of correct programs, in: Semantics of Programming Languages (Proc.Int.Workshop, Bad Honnef, March 1977), Abteilung Informatik, Universität Dortmund, Bericht Nr.4,1 (1977) 25-29.

Blikle, A.(1977B) Towards mathematical structured programming, In: Formal Description of Programming Concepts (Proc.IFIP Working Conf.St.Andrews, N.B. Canada, August 1-5, 1977, E.J. Neuhold, ed.) 183-202, North Holland, Amsterdam 1978.

Blikle, A.(1977C) A comparative review of some program-verification methods. Mathematical Foundations of Computer Science 1977 (Proc.6th Symp.Tatranska Lomnica 1977) Lecture Notes in Computer Science 53 Springer Verlag, Heidelberg 1977, 17-33.

Blikle, A.(1978) Specified programming, In: Mathematical Studies of Information Processing (Proc.Int.Conf.Kyoto,August 1978).

Blikle, A. (1979) A survery of input-output semantics and program verification. ICS PAS Reports 344 (1979).

Burstall, R.M. and Darlington, J. (1977) A transformation system for developing recursive programs, J.ACM, 24(1977), 44-67.

Darlington, J. (1975) Applications of program transformation to program synthesis, Proc.Symp.of Proving and Improving Programs, Arc-et-Senans 1975, pp.133-144.

Darlington, J.(1976) Transforming specifications into efficient programs. New Directions in Algorithmic Languages 1976 (ed. S.A.Schuman), IRIA Rocquencourt 1976.

Dershowitz, N. and Manna, Z. (1977) Inference rules for program annotation, Report No STAN-CS-77-631 (1977).

Dijkstra, E.W. (1968) A constructive approach to the problem of program correctness. BIT 8(1968), 174-186.

Dijkstra, E.W. (1975) Guarded commands, non-determinizm and a calculus for the derivation of programs, Proc.1975, Int. Conf.Reliable Software, pp.2.0-2.13 Also in: Comm.ACM, 18 (1975), 453-457.

Dijkstra, E.W. (1976) Formal techniques and sizable programs, Proc. 1st Conf.Eur.Coop.Inf. Amsterdam 1976, Lecture Notes Comput. Sci. 44, 225-235 (1976).

Emden van, M.H. (1975) Verification conditions as representations for programs. manuscript (1975).

Emden van, M.H. (1976) Unstructured systematic programming. Dept. CS, Univ.Waterloo, CS-76-09 (1976).

Erig, H.; Kreowski, H.J.; Padawitz, P. (1978) Stepwise specification and implementation of abstract data types, in: Automata Languages and Programming (Proc.Fifth Coll.Udine, July 1978), Springer-Verlag LNCS 62, New York 1978, 205-226.

Floyd, R.W. Assigning meanings to programs, Proc.Sym. in Applied Math., 19 (1967), 19-32.

Goguen, J. (1978) Some ideas in algebraic semantics, (manuscript) presented at IBM Conference in Koto (Japan), 1978.

Goguen, J.A.; Thatcher, J.W.; Wagner, E.G.; Wright, J.B. (1975) Abstract data types as initial algebras and correctness of data representations, Proc.Conf.on Comp.Graphics, Patern Recognition and Data Structure, May 1975, 89-93.

Guttag, J. (1977) Abstract data types and the development of data structures. Comm.ACM, 20(1977), 396-404.

Irlik, J. (1976) Constructing iterative version of a system of recursive procedures, In: MFCS (Proc.Int.Symp.MFCS'76) Lecture Notes in CS, Springer-Verlang, Heidelberg 1976, vol.45.

Lee, S.; de Roever, W.R.; Gerhart, S.L. The evolution of list-copying algorithms, Six ACM Symposium on Principles of Programming Languages, January 1979.

Liskov, B.H.; Zilles, S.N. (1975) Specification techniques for data abstraction, IEEE Trans.on SE Vol Se-1 No 1 (1975), 7-19.

Manna, Z. (1974) Mathematical Theory of Computation, Mc Graw-Hill, New York 1974.

Manna, Z.; Pnueli, A. (1974) Axiomatic approach to total correctness of programs, Acta Informatica (1974).

McCarthy, J. A basis for a mathematical theory of computation. In: Computer Programming and Formal Systems, R.Braffort and D.Hirschberg edb., North Holland Amsterdam 1967, pp.33-70.

Spitzen, J.M.; Levitt, K.N.; Lawrence, R. (1976) An example of hierarchial design and proof. New Directions in Algorithmic Languages 1976 (ed. S.A.Schuman), IRJA, Rocquencourt 1976.

Wegbreit, B. (1976) Goal-directed program transformations, IEEE.TSE. Vol SE-2, No 2, (1976), 69-79.