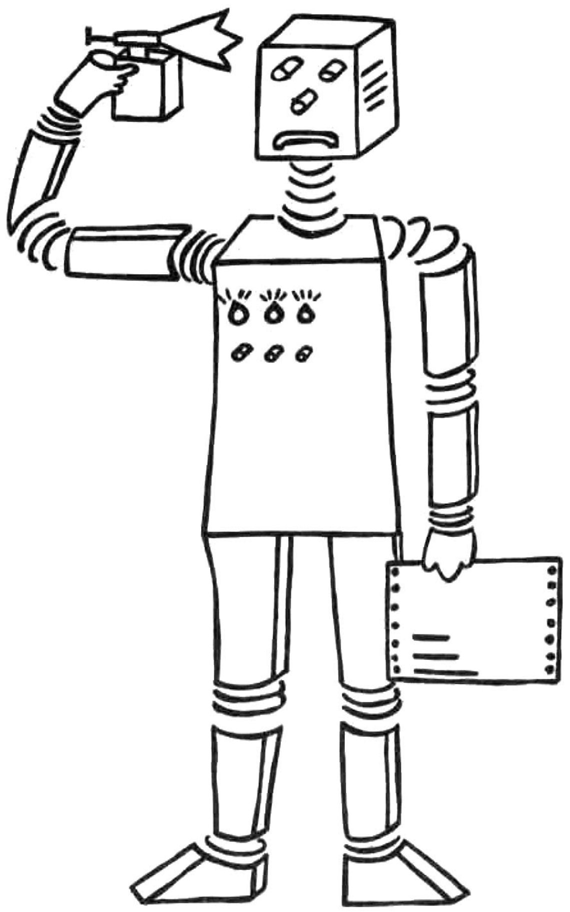
SEKI-REPORT

Artificial
Intelligence
Laboratories

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany



Verification of COBOL Programs

Rolf Socher

August 1986 SEKI-REPORT SR-86-11

VERIFICATION OF COBOL PROGRAMS

Rolf Socher

Universität Kaiserslautern
Fachbereich Informatik
Postfach 3049
6750 Kaiserslautern
F.R. Germany

Abstract:

The use of COBOL for program verification leads to some special problems enforcing the restriction of a verification system to a small subset of the language. This report describes the language Ass Cobol, a subset of the COBOL standard from 1974. The semantic of this language is given by translations of the language constructs into PASCAL statements. Based on this semantic, a system of Hoare-style inference rules is constructed and its correctness is shown.
Two examples demonstrate the complexity of the correctness proofs even of small Ass Cobol programs.

1. Introduction

In this paper we will describe a method for the Hoare-style verification of formally specified COBOL programs. The language being accepted by the system is a subset of the ANSI standard from 1974, enriched with an assertion language and a structuring concept. We will call this language Asserted Cobol (Ass Cobol). The use of COBOL as input language should be justified by its wide application and high standardization, but it implies also some peculiarities enforcing the restriction of the accepted language to a subset of COBOL.

One of the characteristics of COBOL is its large size making it impossible to verify all COBOL constructs. The language design is organized into a kernel, the so called nucleus and some extensions. We restricted the input language Ass Cobol to an essential subset of the nucleus, which contains the most important features of the language.

Another feature of COBOL are some ´dirty´ language constructs which would render verification very difficult. These are e.g. uncontrolled jumps, dynamic alteration of program code and implicit type transformations. These constructs are not allowed in Ass Cobol.

The language COBOL was designed primary for commercial use. Hence some of the main issues of COBOL are input/output and string manipulation. But the very power of Hoare-style program verification lies in the representation of the logic structure of computer programs. Hence the questions of formatting, input/output and string manipulation are not completely treated by our COBOL verifier. Perhaps problems of this kind could be solved in a more elegant (and cheaper) way by code inspection and testing than by program verification.

So why then should we be interested in the verification of COBOL programs anyway? The answer is somehow similar to the "apologia" in the FORTRAN Verification report [BM 80], where the authors spend considerable effort on the justification of the choice of such a "dirty" language. The same arguments apply here: COBOL is certainly a language that does not live up to present day software engineering standards, it is even less suited for formal verification and in fact should have been replaced by more advanced programming languages years ago. But it was not! A lot of business applications are still programmed in COBOL and they will also be in the next future. Another argument for the use of COBOL are some important security issues in the area, where COBOL is mostly used, like access rights on data bases.

The semantics of COBOL is nowhere formally defined. For our purposes we have transformed the informal semantics description of the ANSI report into equivalent WP or PASCAL constructs. WP (While Programs) is a programming language defined in [Ba 80], which consists of some of the most common programming language constructs (see 2.). The axiomatic semantics of PASCAL is described in [HW 73].

2. Notation:

N	is the set of natural numbers
x,y,z	integer variables
a,b,c	string variables
i,j	index variables
a[i]	is the i-th element of the array a (array selection)
r.b	is the record selector b of the record r
r	record variables
k,m,n	constants
L	labels
t	terms
$\lambda(y)$	length of the integer variable x (i.e. number of digits)
$\mu(a)$	length of the string variable a

$\gamma(x)$	upper bound of the array x
var(p)	is the set of all variables occuring in p (a variable x does not occur in the term $\lambda(x)$.)
free(p)	is the set of all variables occuring free in p
p[t/x]	is the result of substituting in p all occurences of x by t

WP is a programming language with the constructs:

$y{:=}x$	assignment
$S_1;S_2$	concatenation
if b then S_1 else S_2 fi	alteration
while b do S od	while

These constructs have the follwing axiomatic definitons:

$$\{p[y/x]\} \quad x:=y \quad \{p\}.$$

$$\frac{\{p\}\ S_1\ \{r\};\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;S_2\ \{q\}}$$

$$\frac{\{p \wedge b\}\ S_1\ \{q\};\ \{p \wedge \neg b\}\ S_2\ \{q\}}{\{p\}\ \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}\ \{q\}}$$

$$\frac{\{p \wedge b\}\ S\ \{p\}}{\{p\}\ \text{while } b \text{ do } S \text{ od}\ \{p \wedge \neg b\}}$$

3. Syntax of Ass-Cobol

The language Ass Cobol consists of the language Cobol and the assertion language Ass.

3.1. The Language Cobol

The language Cobol is a subset of the standard version of the ANSI from 1974 [ANSI 74]. An ANSI-COBOL implementation consists of modules of the following list:

- Nucleus
- Table Handling
- Sequential I/O
- Relative I/O
- Indexed I/O
- Sort-Merge
- Report Writer
- Segmentation
- Library
- Debug
- Inter-Program Communication
- Communication

From this list all modules except the first three are optional. These modules all have up to two levels.

3.1.1 The Block Structure

Ass Cobol has a block structuring concept to facilitate verification. This concept provides the following features [Flo 74]:
A COBOL program consists of units, the so called program components which have the structure

```
(Component name SECTION.)
sequence of statements
EXIT (PROGRAM) resp. STOP RUN.
sequence of paragraphs describing so called elementary components
```

Hence the possible components are

- the main program which is terminated by STOP RUN. Formally it is a section or a sequence of paragraphs.

- a procedure which may be called by CALL. Such a procedure is terminated by EXIT PROGRAM. It is a nonrecursive procedure with call-by-reference parameters. Global variables are not allowed within this kind of procedure. These procedures are compiled separately from the main program. They have the same formal structure as a main program except the USING clause and the LINKAGE SECTION. We name this kind of procedure call-procedure.

- a procedure which is called by PERFORM. This kind of nonrecursive procedure has no parameters and communicates with the main program

Formally it is a section which is terminated by EXIT.

- an elementary component. This is a single paragraph, which may be called by PERFORM. A single paragraph is terminated by the following paragraph name.

The two last kinds of procedures are called perform-procedures.

We have added to ANSI COBOL the following restrictions concerning the block structure:

- A component always has a main part which comprises the beginning of the component up to STOP RUN resp. EXIT PROGRAM or EXIT. This STOP RUN/EXIT PROGRAM/EXIT is the only one in the component.

- Jumps may not lead out from a component, hence they are not allowed at all in elementary components.

In the following we describe the most important syntactic differences between Ass Cobol and the kernel of ANSI COBOL.

3.1.2 Nucleus

In Identification division and environment division there are no restrictions.

3.1.2.1 Data Division

Following clauses are not allowed:
- USAGE IS
- SIGN IS
- SYNCHRONIZED
- JUSTIFIED
- BLANK WHEN ZERO
- the RENAMES clause on level 66

Following modifications are made
- a conditional variable may not refer to a data item, which is declared as FILLER.
- two variables covering the same storage area by a REDEFINES clause must be of type string (arrays of type string and records having only components of type string are considered as strings).

The most substantial restrictions imposed on ANSI COBOL in the data division relate to the picture clause, which is the syntactic construct to define

the type of the variables. ANSI COBOL provides five base types: alphabetic, alphanumeric, numeric, numeric edited, alphanumeric edited. Ass Cobol allows only the type alphanumeric, which is called in the following ´string´ and the type numeric, which contains the subtypes ´nat´ and ´int´ (real is excluded). Hence only three kinds of picture declarations are allowed:
PIC {IS} X...X (type string), PIC {IS} 9...9 (type natural) and
PIC {IS} S 9...9 (type integer).

3.1.2.2 Procedure Division

3.1.2.2.1 Conditions

Here we have the following restrictions:
- due to syntactic reasons the operands of relations may not be arithmetic expressions
- implicit type conversions are not allowed in relations, i.e. both operands must be either of type numeric or of type string, in which case only the equality relation is admitted.
- the class condition (IS NUMERIC/ALPHABETIC) and also
- the Switch Status Condition is not allowed.

3.1.2.2.2 Statements

The following statements are not allowed:
- ACCEPT x FROM DATE/DAY/TIME
- ALTER
- ENTER
- INSPECT
- STRING
- UNSTRING

On the other statements we have the restrictions:
Implicit type conversion is not allowed in Ass Cobol. Therefore assignments may occur only between variables of the same type.
- The ROUNDED clause for arithmetic statements is omitted, because there is no type real.
- The CORRESPONDING clause for add and subtract statements is not admitted.
- The SIZE ERROR clause is unnecessary due to the overflow checks and therefore omitted.
- Multiple results in arithmetic set and move statements are not allowed (e.g. ADD X TO Y, Z, ...).

nonelementary moves are move statements, where sending or receiving data item are not of a base type. These moves are not allowed, because they can lead to implicit type transformations. But the base type string in Ass Cobol is defined as the type alphanumeric together with all group items (i.e. records), all components of which are of type string. Therefore some kind of nonelementary moves in the sense of ANS-COBOL are elementary moves in the sense of Ass Cobol. Two data items in a MOVE CORR statement correspond only, if they are both elementary items

- PERFORM
  In the variant
  PERFORM proc1 [THRU proc2] VARYING i FROM j BY k UNTIL b
  of the PERFORM statement the appendix AFTER m FROM ... is omitted.
- STOP
  the variant STOP literal is excluded; STOP RUN may appear only once in a main program.

3.1.3 Table Handling Module

With the exception of the multiple assignment to index variables (SET i,j,... TO n) level 1 of ANS is fully realized. Additionally there is included the SEARCH statement from level 2.

3.1.3.1 Data Division

Variable array bounds are not admitted in the OCCURS clause and also the ASCENDING/DESCENDING KEY variant is excluded.
The USAGE IS INDEX clause is not allowed, hence there are no so called index data items.

3.1.3.2 Procedure Division

- SEARCH Statement
  The SEARCH ALL format is excluded and the WHEN condition clause may not be repeated.
- SET Statement
  Multiple assignments are not admitted (SET I,J,... TO M).

3.1.4 Sequential I/O module

Here we have three restrictions to level 1 of ANS (the USE statement, the CODE SET clause and the I/O files).

3.1.4.1 Data Division

Following clause is not allowed in the file description:
- CODE SET

3.1.4.2 Procedure Division

- OPEN
  I/O file is excluded
- REWRITE
  relates to I/O files and is therefore omitted
- USE
  this statement (declaration of an error procedure) is not admitted.

3.2. The Language ASS

ASS is the language of the assertions. It is a first order predicate logic language admitting quantifiers. Its syntax is the syntax of COBOL conditions enriched with the keywords assert, invariant, entry, exit, precondition, postcondition, let and initial. It additionally includes:
- undefined functions and predicates and
- predefined functions and predicates (e.g. the predicates true, false, opened and the functions first, rest).

The language ASS is not a subset of COBOL, whence the assertions must be read over by the ANS-COBOL compiler. Therefore they are formally comments.

The language ASS has the following clauses:

- assert
  An assert clause must be added to each paragraph, to which a GOTO statement in the program text jumps. Assert clauses may occur additionally everywhere in the program text, where COBOL statements might occur.

- invariant
  Each loop must contain an invariant. Loops can be constructed with the following statements:
  PERFORM ... TIMES, PERFORM ... UNTIL, PERFORM ... VARYING ... UNTIL.

- entry und exit

These clauses are optional. They denote precondition resp. postcondition for a perform-procedure.

- precondition und postcondition
  These clauses denote precondition resp. postcondition for a main program or a call-procedure

- let
  The let statement is the assignment to logic variable (i.e. a variable not occuring in the COBOL program but only in assertions).

- initial
  This is also an assignment to a logic variable. It defines initial values for procedure parameters (in the case of call-procedures) and for global variables (in the case of perform-procedures).
  The let and the initial statement have a special syntax:
  let $x' = x$ bzw. initial $x' = x$.

4. Data Structures

4.1 Index

Syntax: Declaration of the index variable i within the array declaration
OCCURS n TIMES INDEXED BY i.

Variables of type index obey the axioms of the natural numbers.

4.2 Numeric (nat and int)

Syntax:

x PIC S9(n) for int and
x PIC 9(n) for nat.

x is a variable with at most n decimal digits; in the following the upper bound of the variable x is denoted by $\lambda(x)$ i.e. $\lambda(x) = 10^{**}n-1$. The axioms of natural (resp. integer) numbers are valid for numeric variables. Additionally we have a function cut : $N' \times N \to N'$, together with the definitions

$P(\text{undef}) \equiv \text{false}$ for every predicate $P(x)$ admissible in ASS

$\text{cut}(x,n) = \text{if } x \le n \text{ then } x \text{ else undef}.$

Hence

$P(\text{cut}(x,n)) \equiv \text{if } x \le n \text{ then } P(x) \text{ else false} \equiv x \le n \wedge P(x).$

4.3 Character

The type character does not exist in COBOL. Here it serves only for defining the type string. The elements of type character are 26 letters, 10 digits, the blank character (denoted by ´ ´) and possibly some other elements of the ASCII character set.

4.4 Strings

Syntax: a PIC X(n)
where n is the length of a, which we denote by $\mu(a)$.
The type string may be defined as array of character with the length n. The only relation on strings is the equality relation, which is defined in the following way: Let $\mu(a)=n$, $\mu(b)=m$ and $m \geq n$. Then

$$a = b \Leftrightarrow \forall i\ 1 \leq i \leq n \Rightarrow a(i) = b(i) \wedge \forall j\ n < j \leq m \Rightarrow b(j) = ´\ ´.$$

Additionally we have a substring selection [,], so that a[j,k] is a string with

$$(a[i,j])\,[k] = \begin{cases} a[i+k-1] & \text{if } 1 \leq k \leq j-i+1 \\ \\ ´\ ´ & \text{otherwise} \end{cases}$$

and a function $\langle\ ,\ ,\ \rangle$: string×(nat×nat)×string → string (which may be viewed as a substring modification function) with

$$\langle a,(i,j),e\rangle[k] = \begin{cases} a[k] & \text{if } 1 \leq k < i \text{ or } j < k \leq n \\ e[k-i+1] & \text{if } i \leq k \leq j \\ ´\ ´ & \text{otherwise} \end{cases}$$

These functions are useful to handle group items which are strings and the move statement for string variables.

Example:

```
01    a.
 02   a1   PIC x(n)
 02   a2   PIC x(m) ...
```

Then the parser performs the following transformations:

$a_1 \rightarrow a[1,n]$

$a_2 \rightarrow a[n+1,n+m]$ ...

4.4 Arrays

Syntax: x ... OCCURS n TIMES [INDEXED BY i]
The function $\langle\ ,\ ,\ \rangle$: array×nat×exp → exp is introduced with the following definition (exp is the set of COBOL arithmetic expressions):

$$\langle a,[i],s\rangle\,[j] = \begin{cases} s & \text{if } i=j \\ a(j) & \text{otherwise} \end{cases}$$

We have used the same symbol as for the substring modification function; there is no danger to confuse the two functions.

4.5 Records

Elements of this data type are called group items in ANS-COBOL.
Syntax: If a variable r is declared with a level number $\neq 77$ and no picture clause, it is a record variable. The selectors belonging to r are the consecuting variables with higher level numbers than r.

In an analogous way to arrays (and again with the same symbol) the function $\langle\ ,\ ,\ \rangle$ is defined by

$$\langle r,.i,s\rangle\ .j = \begin{cases} s & \text{if i and j are syntactically equal} \\ r.j & \text{otherwise} \end{cases}$$

4.6 Other Clauses in the Data Division

- the condition name clause
  The condition name clause defines a variable, which represents a condition. The verification system substitutes every occurence of the defined variable in the procedure division by the according condition.

- the value is clause
  this clause is viewed as a assignment

- the redefines clause
  this clause is admitted only for strings, since otherwise it would cause implicit type transformations. The verification system substitutes every occurence of the redefining variable in the procedure division by the according redefined variable.

5. An Axiom System for Programming Language Constructs

In this section we define a semantics for Ass Cobol by giving to each Cobol construct an equivalent WP or PASCAL construct. This semantic leads to a system of inference rules for Cobol constructs. The rules D0 until D4, D5 and D6 of [Hoa 71] will serve as a basis for proving the soundness of our axiom system (i.e. the axiom of assignment, the rules of consequence, composition, iteration, alternation, substitution and adaptation). For the GOTO statement we took the inference rule of [Ba 80].

5.1.Assignment Operations

- simple MOVE (numeric)

  Syntax: MOVE {x|n} TO y.

  Semantic: Let y be a numeric variable with length $\lambda(y)$. The value of y is undefined after the assignment if $x > \lambda(y)$ (resp. $n > \lambda(y)$) [ANS 74]. Hence we have the following deduction rule:

  $\{p[cut(x, \lambda(y))/y]\}$ MOVE x TO y $\{p\}$

  Due to 4.2 we have

$$p[cut(x, \lambda(y))/y] = \begin{cases} p[x/y] \wedge x \leq \lambda(y) & \text{if } y \in free(p) \\ p & \text{otherwise} \end{cases}$$

  hence $\{p[x/y] \wedge x \leq \lambda(y)\}$ MOVE x TO y $\{p\}$ if $y \in free(p)$ and $\{p\}$ MOVE x TO y $\{p\}$ if $y \notin free(p)$.

- simple MOVE (alphanumeric)

  Syntax: MOVE {a|n} TO b.

  Semantic: Let b be a string variable with the length $\mu(b)$. The above statement assigns to b the value of a (resp. n) cut to the length of b, i.e. it is equivalent to the statement $b := a[1, \mu(b)]$. Hence we have the following inference rule:

  $\{p[a(1, \mu(b))/b]\}$ MOVE a TO b $\{p\}$

- MOVE for arrays (numeric).

  Syntax: MOVE y TO x(i).

  Semantic: Let x be an array of length ν and let each x(i) have the length μ. In the case of $i > \nu$ we define the meaning of the above statement as an error statement with the inference rule

  {true} error {false}

  and in the case of $i \leq \nu$ we use the array modification $\langle a,i,y \rangle$ whence we get the inference rule

  $\{i \leq \nu \wedge p[\langle x,[i],cut(y,\mu)\rangle/x]\}$ MOVE y to x[i] $\{p\}$

  The move statement for arrays of strings may be treated analogously.

- non elementary MOVE

  MOVE for group items:
  Syntax: MOVE r_1 TO r_2.
  This move statement is allowed only for variables of type string, which are group items in the sense of ANS-COBOL. Assignments to variables, which are not of a base type are not allowed.

  MOVE CORR:
  Syntax: MOVE CORR r_1 TO r_2.
  r_1 and r_2 are variables of type record. Let x be a record element of r_1, y a record element of r_2. x and y are said to correspond, iff both are of base type and there exists a sequence $s_1,\dots,s_n$ of record selectors, so that $r_1.s_1. \dots .s_n \equiv x$ and $r_2.s_1. \dots .s_n \equiv y$. To each pair of corresponding variables in r_1 and r_2 a move statement is performed when executing the above MOVE CORR statement.
  This assignment is transformed by the Ass Cobol parser into a sequence (MOVE x_i TO y_i) for each corresponding pair (x_i,y_i).

- SET
  Syntax: SET i TO {j|n}.
  This is the assignment operation for variables of type index with the inference rule

  {p[j/i]} SET i TO j {p}

5.2. Arithmetic operations

- ADD, SUBTRACT, MULTIPLY, DIVIDE, COMPUTE
  Syntax: ADD x TO y.
  ADD x TO y GIVING z.
  analogous for SUBTRACT und MULTIPLY.
  DIVIDE x INTO y.
  DIVIDE x INTO y GIVING z (REMAINDER r).
  DIVIDE x BY y.
  DIVIDE x BY y GIVING z (REMAINDER r).
  COMPUTE y = t.
  Semantic: These statements are equivalent to respectively the following ones:
  COMPUTE y = x+y.
  COMPUTE z = x+y.

COMPUTE y = y/x.
COMPUTE z = y/x . (COMPUTE r = rem(y,x).)
COMPUTE z = x/y.
COMPUTE z = x/y. (COMPUTE r = rem (x,y).)

For the COMPUTE statement we have the following inference rule:

$\{p[cut(t, \lambda(y))/y]\}$ COMPUTE y = t $\{p\}$

- SET ... UP BY , SET ... DOWN BY.
  Syntax: SET i {UP | DOWN} BY n.
  Semantic: This statement is equivalent to
  SET i TO i+n resp SET i TO i-n.

5.3 Control structures

- IF ...THEN and IF ... THEN ... ELSE
  Syntax: IF cond THEN S.
  IF cond THEN S_1 ELSE S_2.
  Semantic:
  The same as the analogous construct in WP. Therefore we have the inference rules: (conditional)

$$\frac{\{p \wedge b\}\ S_1\ \{q\}\ ;\ \{p \wedge \neg b\}\ S_2\ \{q\}}{\{p\}\ \text{IF } b \text{ THEN } S_1 \text{ ELSE } S_2\ \{q\}}$$

and

$$\frac{\{p \wedge b\}\ S\ \{q\}\ ;\ p \wedge \neg b \Rightarrow q}{\{p\}\ \text{IF } b \text{ THEN } S\ \{q\}}$$

- the labelled statement and the GO TO statement
  Syntax: GO TO L.
  Semantic: The same as the corresponding construct in PASCAL; see also [Ba 80]. Let $h \equiv \{p_1\}$ GO TO L_1 {false} $\wedge \ldots \wedge$ $\{p_n\}$ GO TO L_n {false}. Then the inference rule for a labelled statement is:

$$\frac{h \Rightarrow \{p_1\}\ A_1\ \{p_2\}, \ldots, h \Rightarrow \{p_n\}\ A_n\ \{p_{n+1}\}}{\{p_1\}\ L_1.\ A_1 \ldots L_n.\ A_n\ \{p_{n+1}\}}$$

- GO TO DEPENDING ON

Syntax: GO TO $L_1, \ldots, L_n$ DEPENDING ON x.

Semantic: corresponds to the following PASCAL statement

if x=1 then goto L_1 else if x=2 then goto L_2 else ...
... if x=n goto L_n fi ... fi;

Inference rule:

$$\frac{p \wedge (x=1) \Rightarrow p_1, \ldots, p \wedge (x=n) \Rightarrow p_n, \; p \wedge x \notin \{1,\ldots,n\} \Rightarrow q}{\{p\}\ \text{GO TO } L_1, \ldots, L_n \text{ DEPENDING ON } x\ \{q\}}$$

Proof:

We want to prove the correctness of this rule, given the above semantic, for n=1:

Let $p \wedge x=1 \Rightarrow p_1$ and $p \wedge x \neq 1 \Rightarrow q$. From this we may deduce:

$\{p \wedge x=1\}$ GO TO L	$\{q\}$	(goto & consequence)
$\{p\}$ IF x=1 THEN GO TO L	$\{q\}$	(conditional)
$\{p\}$ GO TO L DEPENDING ON x	$\{q\}$	(definition of goto depending)

- PERFORM (PERFORM ... THRU)

Syntax: PERFORM P. resp. PERFORM P_1 THRU P_n.

P is a perform-procedure with the declaration P.S (EXIT), $P_1,\ldots,P_n$ are subsequent perform-procedures with the declarations $P_1.S_1$ (EXIT), ...,$P_n.S_n$(EXIT). In the perform ... times, perform ... until, perform ... varying statements (see below) we only treat the perform P variant, the perform P_1 thru P_n variant can be treated analogously.

Semantic: PERFORM P is the (nonrecursive) call of a parameterless perform procedure with global variables.

There are two possible inference rules for the perform statement. The first one is the replacement rule:

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ \text{PERFORM P}\ \{q\}}$$

and analogously for the PERFROM THRU variant:

$$\frac{\{p\}\ S_1; \ldots; S_n\ \{q\}}{\{p\}\ \text{PERFORM } P_1 \text{ THRU } P_n\ \{q\}}$$

The second inference rule is the rule of adaptation [Hoa71]. One drawback of the the replacement rule, if it is used to generate a verification condition for the procedure call, is the fact that the whole procedure body has to be verified each time the procedure is called. To apply the rule of adaptation, the procedure body is verified only once relative to its entry and exit condition. Therefore the rule of adaptation, which uses the entry and exit condition of the procedure definition, seems to be more appropriate to generate verification conditions, at least in these cases, where a procedure is called more than once. But in the perform ... times, perform ... until, perform ... varying statements we give only inference rules, which are based on the replacement rule.
Let $\underline{a}$ be a list of all variables being set by S (i.e. appearing on the left of an assignment statement in S), $\underline{k}$ a list of all variables occuring free in p and r, but not in $\underline{a}$ or s. The rule of adaptation is:

$$\frac{\{p\}\ \text{P.S (EXIT)}\ \{r\}}{\{\exists \underline{k}\ (p \wedge \forall \underline{a}\ (r \Rightarrow s))\}\ \text{PERFORM P}\ \{s\}}$$

- PERFORM ... TIMES.

Syntax: PERFORM P {x|n} TIMES.

Semantic: Corresponds to the following WP statement:

$i:=0;$ while $i \neq x$ do S ; $i:= i+1$ od.
with $\{i\} \cap \text{var}(S,x) = \emptyset$.

Inference rule:

$$\frac{p \Rightarrow r[0/i];\ \{r\}\ S\ \{r[i+1/i]\};\ r[x/i] \Rightarrow q}{\{p\}\ \text{PERFORM P x TIMES}\ \{q\}}, \quad \text{if } \{i\} \cap \text{free}(p,q,S) = \emptyset.$$

Proof:
Let $p \Rightarrow r[0/i]$, $\{r\}\ S\ \{r[i+1/i]\}$ and $r[x/i] \Rightarrow q$. Then we have the inferences

$\{r \wedge i \neq x\}$ $S;i:=i+1$	$\{r\}$	(assignment, composition, consequence)
$\{r\}$ while $i \neq x$ do $S;i:=i+1$ od	$\{r \wedge i=x\}$	(iteration)
$\{p\}$ $i:=0;$ while $i \neq x$ do $S;$ $i:=i+1$ od	$\{q\}$	(consequence)
$\{p\}$ PERFORM P x TIMES	$\{q\}$	(definition of perform ... times)

- PERFORM ... UNTIL.
  Syntax: PERFORM P UNTIL cond.
  Semantic: the same as the PASCAL construct repeat S until cond, whence we have the inference rule [HW 73] :

$$\frac{\{p\}\ S\ \{q\}\ ;\ q \wedge \neg b \Rightarrow p}{\{p\}\ \text{PERFORM P UNTIL } b\ \{q \wedge b\}}$$

- PERFORM ... VARYING... UNTIL
  Syntax: PERFORM P VARYING x FROM {y|n} BY {z|m} UNTIL cond.
  Semantic: Corresponds to the following PASCAL statement

  x := y; repeat S; x:= x+z until cond;

  Inference rule:

$$\frac{\{p\}\ S\ \{q[x+z/x]\}\ ,\ q \wedge \neg b \Rightarrow p}{\{p[y/x]\}\ \text{PERFORM P VARYING } x \text{ FROM } y \text{ BY } v \text{ UNTIL } b\ \{q \wedge b\}}$$

  Proof:
  Let {p} S {q[x+z/x]} and $q \wedge \neg b \Rightarrow p$. Then
  {p} S;x:=x+z {q}; $q \wedge \neg b \Rightarrow p$ (assignment & consequence)
  {p[y/x]} x:=y; repeat S;x:=x+z until b {$q \wedge b$} (repeat & assignment & consequence)
  {p[y/x]} PERFORM P VARYING x FROM y BY z UNTIL b {$q \wedge b$} (definition)

- CALL ... USING
  Syntax: CALL proc USING $(x_1 | r_1),...,(x_n | r_n)$
  Semantic: The call ... using statement is the (nonrecursive) call of a procedure with call-by-reference parameters and without global variables. Let S be the body of the procedure proc and x̲ the formal parameter list. A premise of the following inference rules is the prohibition of aliasing, i.e. the actual parameters must be syntactically different and must also occupy different storage areas. Otherwise we could construct for the the procedure p(x y) with the body COMPUTE x = y+1 the following inference:

  {true} COMPUTE x = y+1. {x=y+1} (assignment)
  {true} CALL proc USING x y {x=y+1} (1.)

$\{true\}$ CALL proc USING a a $\{a=a+1\}$ (2.)
which is a contradiction.

Analogous to the perform statement we have the choice between a replacement rule and a rule of adaptation. If we want to use the replacement rule, we need additionally two substitution rules, a parameter-substitution-rule and a variable-substitution-rule:

1. Procedure-call-rule:

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ \text{CALL proc USING}\ \underline{x}\ \{q\}}$$

2. Parameter-substitution-rule

$$\frac{\{p\}\ \text{CALL proc USING}\ \underline{x}\ \{q\}}{\{p[\underline{u}/\underline{x}]\}\ \text{CALL proc USING}\ \underline{u}\ \{q[\underline{u}/\underline{x}]\}}\ ,$$

if $\underline{u} \cap \text{free}\ (p,q) \subseteq \underline{x}$

3. Variable-substitution-rule:

$$\frac{\{p\}\ \text{CALL proc USING}\ \underline{u}\ \{q\}}{\{p[\underline{s}/\underline{z}]\}\ \text{CALL proc USING}\ \underline{u}\ \{q[\underline{s}/\underline{z}]\}}\ , \quad \text{if}\ \underline{s},\underline{z} \notin \text{free}\ (S[\underline{u}/\underline{x}])\ .$$

The variable substitution rule may be necessary to solve name conflicts.
The rule of adaptation [Hoa 71]:
Analogous to the perform procedures we have the following rule, which is, together with the parameter substitution rule, equivalent to the above rules. Let p and r be the pre- resp. postcondition for the procedure P, $\underline{a}$ be the list of actual parameters of P subject to the restrictions mentioned above (prohibition of aliasing), and let $\underline{k}$ be a list of all variables occuring free in p and r, but not in $\underline{a}$ or s (i.e. the initial values of the parameters).

$$\frac{\{p\}\ \text{CALL P USING}\ \underline{a}\ \{r\}}{\{\exists \underline{k}\ (p \wedge \forall \underline{a}\ (r \Rightarrow s))\}\ \text{CALL P USING}\ \underline{a}\ \{s\}}$$

- ENTRY ... USING ... EXIT PROCEDURE.
  Syntax: ENTRY proc USING $\{a_1 \mid r_1\},\ldots,\{a_n \mid r_n\}$.
  Semantic: procedure declaration for a call procedure.
  Inference rule: Let proc be a procedure with body S.

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ \text{ENTRY proc USING } \underline{a}.\ S\ \text{EXIT PROCEDURE}\ \{q\}}$$

- NEXT SENTENCE
  Syntax: occurs in the following statements:
  IF cond THEN {S/NEXT SENTENCE} ELSE {S/NEXT SENTENCE} . and
  SEARCH tab VARYING i WHEN cond {S/NEXT SENTENCE}.
  Semantic: corresponds to the dummy statement
  Inference rule:

$$\frac{p \Rightarrow q}{\{p\}\ \text{NEXT SENTENCE}\ \{q\}}$$

- Procedure declaration ... EXIT
  Declaration of a perform procedure
  Syntax: P. S EXIT.
  Inference rule:

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ P.\ S\ \text{EXIT}\ \{q\}}$$

  where p and q are entry resp. exit condition for the procedure P.

- Program declaration ... STOP RUN
  Syntax: P. S STOP RUN.
  Inference rule:

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ P.\ S\ \text{STOP RUN}\ \{q\}}$$

- SEARCH ... VARYING
  Syntax: SEARCH tab {VARYING i} AT END S WHEN cond_1 S_1 ... WHEN cond_n S_n.
  Tab designates an one dimensional array and i is the corresponding index variable.
  Semantic: Let I be the length of the array tab. Then the following WP statement is equivalent to the above search statement:
  while $(i \le I \wedge \neg\text{cond}_1 \wedge \dots \wedge \neg\text{cond}_n)$ do i:= i+1 od;
  if $i > I$ then S else if cond_1 then S_1 else ... if cond_n then S_n fi ... fi;
  Inference rule: (here for n=1)

(1) $\{p \wedge \forall \kappa : i \leq \kappa \leq I \Rightarrow \neg\, cond[\kappa/i] \wedge j > I\}$ SET i TO j; S $\{q\}$

(2) $\{p \wedge i \leq j \leq I \wedge cond[j/i] \wedge \forall \kappa : i \leq \kappa < j \Rightarrow \neg cond[\kappa/i]\}$ SET i TO j; S_1 $\{q\}$

$\{p\}$ SEARCH tab AT END S WHEN cond S_1 $\{q\}$

where $\{j\} \cap var(p,q,S) = \emptyset$

Proof: We assume (1) and (2). First we show:

(*) $\{ p \wedge \forall \kappa : i \leq \kappa < j \Rightarrow \neg cond[\kappa/i] \}$ while ($j \leq I \wedge \neg cond[j/i]$) do $j := j+1$ od
$\{p \wedge \forall \kappa : i \leq \kappa < j \Rightarrow \neg cond[\kappa/i] \wedge (j > I \vee cond[j/i])\}$

We have

$(p \wedge \forall \kappa : i \leq \kappa < j \Rightarrow \neg cond[\kappa/i] \wedge j \leq I \wedge \neg cond[j/i]) \Rightarrow$
$(p \wedge \forall \kappa : i \leq \kappa < j+1 \Rightarrow \neg cond[\kappa/i])$,

hence (from the assignment axiom)

$\{p \wedge \forall \kappa : i \leq \kappa < j \Rightarrow \neg cond[\kappa/i] \wedge j \leq I \wedge \neg cond[j/i] \}$ $j := j+1$
$\{ p \wedge \forall \kappa : i \leq \kappa < j \Rightarrow \neg cond[\kappa/i] \}$,

and from the while-rule follows (*).
Moreover we have

(**) $p \Rightarrow p \wedge \forall \kappa : i \leq \kappa < i \Rightarrow \neg cond[\kappa/i]$.

From (1) and (2) we deduce with the conditional rule

(***)$\{p \wedge \forall \kappa : i \leq \kappa < j \Rightarrow \neg cond[\kappa/i] \wedge (j > I \vee cond[j/i])\}$
if $j > I$ then $i := j$;S else if $cond[j/i]$ then $i := j$;S_1 fi fi. $\{q\}$

From (*),(**) und (***) follows with the composition rule

$\{p\}$ $j := i$; while ($j \leq I \wedge \neg cond[j/i]$) do $j := j+1$ od;
if $j > I$ then $i := j$;S else if $cond[j/i]$ then $i := j$;S_1 fi fi; $\{q\}$

which is equivalent to

$\{p\}$ while ($i \leq I \wedge \neg cond$) do $i := i+1$ od;
if $i > I$ then S else if cond then S_1 fi fi; $\{q\}$

and from the definition of the search statement we get

$\{p\}$ SEARCH tab AT END S WHEN cond S_1 $\{q\}$.

These formulae will become very complicated in the case of $n>1$, therefore a manageable verification system should allow only $n=1$.

5.4 I/O-Operations:

- ACCEPT/DISPLAY

  Syntax: ACCEPT x . and
  DISPLAY x.

  Semantic: Input/Output via terminal.

  We adopt the method of [Ma 75]. Each occurence of a statement ACCEPT x is followed by an annotation which designates to x an initial value x´, a so called logical variable, which may occur only in the assertions. Analogously each occurence of a statement DISPLAY x is preceeded by a clause which asserts the current value of the output variable x in terms of its initial value x´.

  Example:

  ```
  ACCEPT x.
  { let x´ = x}
  ADD 2 TO x.
  {assert x = 2 + x´}
  DISPLAY x.
  ```

  Inference rules:

  $\{p[x'/x]\}$ let $x' = x$ $\{p\}$ and

  $$\frac{p \Rightarrow r \wedge r \Rightarrow q}{\{p\} \text{ assert } r \ \{q\}}$$

- sequential files

  Syntax: There are two kinds of sequential files in COBOL: input-files, where only the operations OPEN INPUT file, CLOSE file and READ file AT END S are allowed and output-files, where only OPEN OUTPUT file, CLOSE file and WRITE record are allowed.

  Semantic: A sequential file is described as a sequence $\langle f_1, \ldots, f_n \rangle$ of data records; $\langle\rangle$ denotes the empty file ([AA 78],[HW 73]). The data records are all of the same type T.

The input file f is given by a pair (f_L, f_R) of type $T^* \times T^*$ (with the intuitive meaning: f_L is the sequence of all records which have been read, f_R is the rest of the file) and a buffer variable $f\uparrow$ of typ T. An output file is given by a variable f_L of type T^* and a buffer variable $f\uparrow$ of type T. Besides we have the functions first, rest with their obvious meanings, the predicate eof, which is defined by

$$\mathrm{eof}(f) \equiv f_R = \langle\rangle$$

and the predicate opened. The latter allows to detect programming errors like multiple opening of files, access to files, which have not yet been opened etc.

$\circ$ denotes the concatenation of files.

The standard procedures may be specified in the following manner:

The statement READ file {INTO x} AT END S is transformed into

IF NOT eof(file) CALL read(file); {MOVE $f\uparrow$ TO x} ELSE S.

with a procedure read which may be specified by:

entry condition: $\{\mathrm{opened}(f) \wedge f'=f\}$

exit condition: $\{\mathrm{opened}(f) \wedge f\uparrow=\mathrm{first}\,(f'_R) \wedge f_L=f'_L\circ\langle\mathrm{first}(f'_R)\rangle \wedge f_R=\mathrm{rest}\,(f'_R)\}$

The statement WRITE $f\uparrow$ {FROM x} is transformed to

{MOVE x TO $f\uparrow$;} CALL write (file)

where the procedure write has the specification

entry condition: $\{\mathrm{opened}(f) \wedge f'=f\}$

exit condition: $\{\mathrm{opened}(f) \wedge f_L=f'_L\circ\langle f\uparrow\rangle$,

The open statement changes the opened-status of the file and assigns initial values to the f_L and f_R variables. It is specified by:

entry condition: $\{\neg\ \mathrm{opened}(f) \wedge f'=f\}$

exit condition: $\{\mathrm{opened}(f) \wedge f_L=\langle\rangle \wedge f_R=f'\}$ (for input files)

exit condition: $\{\mathrm{opened}(f) \wedge f_L=\langle\rangle\ \}$ (for output files)

For the procedure close we have the specification

entry condition: $\{\mathrm{opened}(f)\}$

exit condition: $\{\neg\ \mathrm{opened}(f)\}$

5.5 Constructs which are not verified

Some frequently occuring ANS-COBOL constructs are not part of Ass Cobol, because they violate rules of structured programming or they cannot be verified within the Hoare calculus.

5.5.1.Statements

- ALTER
  This is a construction, which alters dynamically program code. This construct will not be part of the future ANS standard language and it can not be recommended for a good programming style [Flo 74].

- STRING, UNSTRING, INSPECT;
  These statements perhaps will be treated in future versions of the verification system.

- All operations, which would lead to assignments of variables of different types:
  MOVE for group items (except strings)
  REDEFINES for variables (except strings)

5.5.2. Data structures

- Real numbers
  Until now the arithmetic of real numbers could not be treated by a program verification system.

- The numeric edited and alphanumeric edited type and some clauses controlling ouput formats.
  These are clauses like BLANK WHEN ZERO, JUSTIFIED etc. Since we concentrate on the verification of the control structures we do not consider these clauses.

5.5.3.Conditions

- The condition x IS (NUMERIC|ALPHABETIC), which gives true, if the value of x is numeric (alphabetic).

6. Rules for the VCG

The following inference rules for COBOL constructs are formulated in such a manner, that they generate subgoals from a formula {p}S {q} unambigously. This means, that no two conclusions may have common instances. For that reason the composition rule is not formulated explicitly, but it is contained implicitly in the other rules. From these rules a recursive procedure generating verification conditions may be constructed. Most of these rules may be found in [ILL 73].

(C1) $$\frac{\{p\}\ S\ \{q[t/x] \wedge t \leq \lambda(x)\}}{\{p\}\ S;\ \text{move } t \text{ to } x\ \{q\}}\ , \text{ if } x \in \text{free}(q)\ ,\ x \text{ numeric}$$ (simple assignment)

(C1′) $$\frac{\{p\}\ S\ \{q\}}{\{p\}\ S;\ \text{move } t \text{ to } x\ \{q\}}\ , \text{ if } x \notin \text{free}(q)\ ,\ x \text{ numeric}$$ (simple assignment)

(C1′′) $$\frac{\{p\}\ S\ \{q[t(1,\mu(x))/x]\ \}}{\{p\}\ S;\ \text{move } t \text{ to } x\ \{q\}}\ , \text{ if } x \notin \text{free}(q)\ ,\ x \text{ string}$$ (simple assignment)

(C2) $$\frac{\{p\}\ S\ \{q[\langle x,i,t\rangle/x\,] \wedge i \leq \gamma(x) \wedge t \leq \lambda(x)\}}{\{p\}\ S;\ \text{move } t \text{ to } x(i)\ \{q\}}\ , \text{ if } x \in \text{free}(q)$$ (array assignment)

(C2′) $$\frac{\{p\}\ S\ \{q \wedge i \leq \gamma(x)\}}{\{p\}\ S;\ \text{move } t \text{ to } x(i)\ \{q\}}\ , \text{ if } x \notin \text{free}(q)$$ (array assignment)

and analogous for strings.

(C3) $$\frac{p \Rightarrow q}{\{p\}\ \text{dummy}\ \{q\}}$$ (consequence)

(C3′) $$\frac{\{p\}\ S\ \{q \Rightarrow r\}}{\{p\}\ S;\ \text{q-if}\ \{r\}}$$

q-if is an assertion

(C3´´) $$\frac{\{p\}\ S\ \{q\};\ q \Rightarrow r}{\{p\}\ S;\ q\ \{r\}}$$ q is an assertion

(C4) $$\frac{\{p\}\ S;b\text{-if};S_1\{q\};\ \{p\}\ S;\neg b\text{-if};S_2\{q\}}{\{p\}\ S;\ \text{if } b \text{ then } S_1 \text{ else } S_2\ \{q\}}$$ (conditional)

(C5) $$\frac{\{p\}\ S\ \{\text{assertion at } L\}}{\{p\}\ S;\ \text{goto } L\ \{q\}}$$ (goto)

(C6) $$\frac{\{p\}\ S\ \{x=1 \Rightarrow p_1\};\ldots;\ \{p\}\ S\ \{x=n \Rightarrow p_n\};\quad \{p\}\ S\ \{x \notin \{1,\ldots,n\} \Rightarrow q\}}{\{p\}\ S;\ \text{goto } L_1,\ldots,L_n \text{ depending on } x\ \{q\}}$$ (goto depending)

(p_i is the assertion at L_i)

in C7 until C9 let P be a paragraph with declaration P.R and $\underline{a}$ a list of all global variables of P.

(C7) $$\frac{\{p\}\ S\ \{r \wedge \forall \underline{a}\ (s \Rightarrow q)\}}{\{p\}\ S;\ \text{perform } P\ \{q\}},$$

{ r}, { s} is entry resp exitcondition for P

(C8) $$\frac{\{p\}\ S\ \{r[0/i]\};\ \{r \wedge i<x\}\ R\ \{r[i+1/i]\};\ r[x/i] \Rightarrow q}{\{p\}\ S;\ r;\ \text{perform } P\ x \text{ times}\ \{q\}}$$ (perform...times)

i is a new variable with $\{i\} \cap var(p,q,S) = \emptyset$ and r an assertion

(C9) $$\frac{\{p\}S;\ R\ \{r \wedge \forall \underline{a}(r \wedge b \Rightarrow q)\};\ \{r \wedge \neg b\}\ R\ \{r\};}{\{p\}\ S;\ r;\ \text{perform } P \text{ until } b\ \{q\}}$$, r is an invariant. (perform...until)

(C10) $$\frac{\{p\}S;\ \text{move } y \text{ to } x;\ R\ \{r[x+z/x] \wedge x+z \leq \lambda(x)\};\quad \{r \wedge \neg b\}\ R\ \{r[x+z/x] \wedge x+z \leq \lambda(x)\};\ r \wedge b \Rightarrow q}{\{p\}\ S;\ r;\ \text{perform } P \text{ varying } x \text{ from } y \text{ by } z \text{ until } b\ \{q\}}$$ (perform...varying)

(C11) $$\frac{\{p\}\ S\ \{(r[\underline{a}/\underline{x}] \wedge \forall \underline{z}\ (s[\underline{z}/\underline{x}] \Rightarrow q[\underline{z}/\underline{a}]\ \}}{\{p\}\ S,\ \text{call proc using } \underline{a}\ \{q\}}$$ (procedure call)

proc is a procedure with {entry r} proc($\underline{x}$) {exit s} and all the actual parameters a_i are pairwise different.

(C12) $$\frac{\{p\}\ S\ \{q\}}{\{p\}\ \text{entry proc using } x,\ S,\ \text{return}\ \{q\}}$$ (procedure declaration)

We have the following modifications to [ILL 73]:
The simple assignment rule has to consider the case of overflow;
the array assignment has to consider the violation of array bounds;
the goto depending statement;
the perform times, perform varying and perform until statement.
The perform-until statement requires a word of explanation. We did not choose the following more obvious rule:

(C9') $$\frac{\{p\}\ S;R\ \{r\}\ ;\ \{r \wedge \neg b\}\ R\ \{r\}\ ;\ r \wedge b \Rightarrow q}{\{p\}\ S\ ;\ r\ ,\ \text{perform P until b}\ \{q\}}$$

because this rule requires the invariant r together with b to contain all the information, which is necessary to deduce q. But we want r to contain only the information necessary to describe the loop invariant (see example 1). The rule C9 is an obvious application of the rule of adaptation, as it is formulated in [Ol 83].

7. Examples

The first example shows the application of rule C9. For the sake of simplicity we do not consider the overflow conditions, this will be made in detail in example 2.

Example 1:
Consider the verification task:

```
{z=b ∧ a>0}
move 0 to y;
move a to x;
perform P (invariant x+y=a) until x=0.
{y=a ∧ z=b}.
```
where P is defined by:
```
P. subtract 1 from x;
   add 1 to y.
```

Obviously the invariant x+y=a describes the loop completely. But with rule C9′ we cannot deduce, that z be b after execution of the loop. So we had to add z=b to the invariant. But if we use rule C9, we do not need the additional clause z=b:

We apply C9 to get

```
(1) {z=b ∧ a>0}
     move 0 to y;
     move a to x;
     subtract 1 from x;
     add 1 to y
     {x+y=a ∧ ∀x,y (x+y=a ∧ x=0 ⇒ y=a ∧ z=b)}      and

(2) {x+y=a ∧ ¬a=0}
     subtract 1 from x;
     add 1 to y
     {x+y=a}
```

(2) is proved by two applications of C1 and (1) yields (after some applications of C1 and arithmetical simplifications):

(3) $z=b \wedge a>0 \Rightarrow a-1+1=a \wedge z=b \wedge \forall x,y\ (x+y=a \wedge x=0 \Rightarrow y=a)$,
which can be proved easily.

Example 2:

The following example program calculates quotient and remainder of two integer numbers with the Euclidean algorithm. With this example we want to demonstrate the complications which arise when the overflow checks have to be performed. This example is formulated as a procedure with formal parameters a,b,q and r and local variables x and y. Besides the conditions which ensure the formal correctness of the algorithm we have to find the conditions, which ensure that no overflow will take place for any input

variables a and b satisfying the entry condition. These conditions have to relate the lengths of the program variables. If we take for example a procedure, which exchanges the values of two variables x and y, we surely would get the condition, that the lengths of x and y should be equal.
This example shows that in all assertions of the kind $P(\underline{x})$ we have to add the clause $\underline{x} \le \lambda(\underline{x})$.

It also shows that the overflow checks make the generation and the simplification of the verification conditions very complicated. Therefore a practicable verification system should offer the option of suppressing the overflow checks for a subset of the variables.

```
      entry 'euclid' using a b q r.
      { entry a≥0 ∧ b>0 ∧ a≤λ(a) ∧ b≤λ(b)}
      move a to x.
      move 0 to y.
L1.   { yb+x = a ∧ b>0 ∧ 0≤x ∧ x≤λ(x) ∧ y≤λ(y) ∧  a≤λ(a) ∧ b≤λ(b) }
      if x≥b then subtract b from x; add 1 to y; goto L1.
      move y to q.
      move x to r.
      return.
      { exit qb+r = a ∧ 0≤r<b ∧ q≤λ(q) ∧ r≤λ(r)}
```

First of all we can only apply C12 leading to the subgoal

```
(1)   { a≥0 ∧ b>0 ∧ a≤λ(a) ∧ b≤λ(b) }
      move a to x.
      move 0 to y.
      L1. { yb+x = a ∧ b>0 ∧ 0≤x ∧ x≤λ(x) ∧ y≤λ(y) ∧  a≤λ(a) ∧ b≤λ(b)  }
      if x≥b then subtract b from x; add 1 to y; goto L1.
      move y to q.
      move x to r.
      { qb+r = a ∧ 0≤r<b ∧ q≤λ(q) ∧ r≤λ(r)}
```

Then we get:

```
(2)   { a≥0 ∧ b>0 ∧ a≤λ(a) ∧ b≤λ(b) }                       C1 / from (1)
      move a to x.
      move 0 to y.
      L1. { yb+x=a ∧ b>0 ∧ 0≤x ∧ x≤λ(x) ∧ y≤λ(y) ∧ a≤λ(a )∧ b≤λ(b)}
      if x≥b then subtract b from x; add 1 to y; goto L1.
```

```
        move y to q.
        $\{ qb+x = a \land 0 \le x < b \land q \le \lambda(q) \land x \le \lambda(r) \}$

(3)     $\{ a \ge 0 \land b > 0 \land a \le \lambda(a) \land b \le \lambda(b) \}$            C1 / from (2)
        move a to x.
        move 0 to y.
        L1. $\{ yb+x=a \land b>0 \land 0 \le x \land x \le \lambda(x) \land y \le \lambda(y) \land a \le \lambda(a) \land b \le \lambda(b) \}$
        if $x \ge b$ then subtract b from x; add 1 to y; goto L1.
        $\{ yb+x = a \land 0 \le x < b \land y \le \lambda(q) \land x \le \lambda(r) \}$

(4)     $\{ a \ge 0 \land b > 0 \land a \le \lambda(a) \land b \le \lambda(b) \}$            C4 / from (3)
        move a to x.
        move 0 to y.
        L1. $\{ yb+x=a \land b>0 \land 0 \le x \land x \le \lambda(x) \land y \le \lambda(y) \land a \le \lambda(a) \land b \le \lambda(b) \}$
         $x \ge b$-if; subtract b from x; add 1 to y; goto L1.
        $\{ yb+x = a \land 0 \le x < b \land y \le \lambda(q) \land x \le \lambda(r) \}$

(5)     $\{ a \ge 0 \land b > 0 \land a \le \lambda(a) \land b \le \lambda(b) \}$            C4 / from (3)
        move a to x.                                      (else missing)
        move 0 to y.
        L1. $\{ yb+x=a \land b>0 \land 0 \le x \land x \le \lambda(x) \land y \le \lambda(y) \land a \le \lambda(a) \land b \le \lambda(b) \}$
        $\neg\, x \ge b$-if
        $\{ yb+x = a \land 0 \le x < b \land y \le \lambda(q) \land x \le \lambda(r) \}$

(6)     $\{ a \ge 0 \land b > 0 \land a \le \lambda(a) \land b \le \lambda(b) \}$            C5 / from (4)
        move a to x.
        move 0 to y.
        L1. $\{ yb+x=a \land b>0 \land 0 \le x \land x \le \lambda(x) \land y \le \lambda(y) \land$
               $a \le \lambda(a) \land b \le \lambda(b) \}$
         $x \ge b$-if; subtract b from x; add 1 to y
        $\{ yb+x=a \land b>0 \land 0 \le x \land x \le \lambda(x) \land y \le \lambda(y) \land$
               $a \le \lambda(a) \land b \le \lambda(b) \}$

(7)     $\{ a \ge 0 \land b > 0 \land a \le \lambda(a) \land b \le \lambda(b) \}$            C3' / from (5)
        move a to x.
        move 0 to y.
        L1. $\{ yb+x=a \land b>0 \land 0 \le x \land x \le \lambda(x) \land y \le \lambda(y) \land$
               $a \le \lambda(a) \land b \le \lambda(b) \}$
        $\{ \neg\, x \ge b \Rightarrow yb+x = a \land 0 \le x < b \land y \le \lambda(q) \land x \le \lambda(r) \}$

(8)     $\{ a \ge 0 \land b > 0 \land a \le \lambda(a) \land b \le \lambda(b) \}$            C3 / from (7)
        move a to x.
```

```
      move 0 to y.
      { $yb+x=a \land b>0 \land 0\le x \land x\le\lambda(x) \land y\le\lambda(y) \land$
            $a\le\lambda(a) \land b\le\lambda(b)$ }

(I)    $yb+x=a \land b>0 \land 0\le x \land x\le\lambda(x) \land y\le\lambda(y) \land$            C3 / from (7)
            $a\le\lambda(a) \land b\le\lambda(b) \land \neg\, x\ge b \Rightarrow$
      $yb+x = a \land 0\le x<b \land y\le\lambda(q) \land x\le\lambda(r)$

(9)   { $a\ge 0 \land b>0 \land a\le\lambda(a) \land b\le\lambda(b)$ }                        C1 / from (6)
      move a to x.
      move 0 to y.
      L1. { $yb+x=a \land b>0 \land 0\le x \land x\le\lambda(x) \land y\le\lambda(y) \land$
            $a\le\lambda(a) \land b\le\lambda(b)$ }
       $x\ge b$-if; subtract b from x
      { $(y+1)b+x=a \land b>0 \land 0\le x \land x\le\lambda(x) \land y+1\le\lambda(y) \land$
            $a\le\lambda(a) \land b\le\lambda(b)$ }

(10)  { $a\ge 0 \land b>0 \land a\le\lambda(a) \land b\le\lambda(b)$ }                        C1 / from (9)
      move a to x.
      move 0 to y.
      L1. { $yb+x=a \land b>0 \land 0\le x \land x\le\lambda(x) \land y\le\lambda(y) \land$
            $a\le\lambda(a) \land b\le\lambda(b)$ }
       $x\ge b$-if
      { $(y+1)b+x-b=a \land b>0 \land 0\le x-b \land x-b\le\lambda(x) \land y+1\le\lambda(y) \land$
            $a\le\lambda(a) \land b\le\lambda(b)$ }

(11)  { $a\ge 0 \land b>0 \land a\le\lambda(a) \land b\le\lambda(b)$ }                        C3′ / from (10)
      move a to x.
      move 0 to y.
      { $yb+x=a \land b>0 \land 0\le x \land x\le\lambda(x) \land y\le\lambda(y) \land$
            $a\le\lambda(a) \land b\le\lambda(b)$ }
      { $x\ge b \Rightarrow (y+1)b+x-b=a \land b>0 \land 0\le x-b \land x-b\le\lambda(x) \land y+1\le\lambda(y) \land$
            $a\le\lambda(a) \land b\le\lambda(b)$ }

(II)  $yb+x=a \land b>0 \land 0\le x \land x\le\lambda(x) \land y\le\lambda(y) \land$            C3″ / from (11)
            $a\le\lambda(a) \land b\le\lambda(b) \Rightarrow$
      $x>b \Rightarrow (y+1)b+x-b=a \land b>0 \land 0\le x-b \land x-b\le\lambda(x) \land y+1\le\lambda(y) \land$
            $a\le\lambda(a) \land b\le\lambda(b)$

(12)  { $a\ge 0 \land b>0 \land a\le\lambda(a) \land b\le\lambda(b)$ }                        C1 / from (8)
      move a to x.
      { $0b+x=a \land b>0 \land 0\le x \land x\le\lambda(x) \land 0\le\lambda(y) \land a\le\lambda(a) \land b\le\lambda(b)$ }
```

(13) $\{ a\geq 0 \wedge b>0 \wedge a\leq\lambda(a) \wedge b\leq\lambda(b) \}$ C1 / from (12)

$\{ 0b+a=a \wedge b>0 \wedge 0\leq a \wedge a\leq\lambda(x) \wedge 0\leq\lambda(y) \wedge$
$a\leq\lambda(a) \wedge b\leq\lambda(b)\}$

(III) $a\geq 0 \wedge b>0 \wedge a\leq\lambda(a) \wedge b\leq\lambda(b) \Rightarrow$ C3 / from (13)

$0b+a=a \wedge b>0 \wedge 0\leq a \wedge a\leq\lambda(x) \wedge 0\leq\lambda(y) \wedge$
$a\leq\lambda(a) \wedge b\leq\lambda(b)$

In our example VCG generates the verification conditions I,II and III.

Simplification of the VC's

(I) The first two clauses of the conclusion follow from the premise with the arithmetical transformation:
$\neg (x\leq y) \Rightarrow (x>y)$
It remains to show:

(1) $y\leq\lambda(y) \Rightarrow y\leq\lambda(q)$ and

(2) $x\leq\lambda(x) \wedge x<b \wedge b\leq\lambda(b) \Rightarrow x\geq\lambda(r)$

With the transitivity of $\leq$ and the formula

$$x\leq\alpha \wedge x\leq\beta \Rightarrow x\leq\min(\alpha,\beta)$$

we get from (2)

(3) $x\leq\min(\lambda(x),\lambda(b)) \Rightarrow x\leq\lambda(r)$

We apply the formula

(*) $$(\forall x\ (x\leq\alpha \Rightarrow x\leq\beta)) \Rightarrow \alpha\leq\beta$$

and get from (1) and (3)

(i) $\lambda(y) \leq \lambda(q)$ and

(ii) $\min(\lambda(x),\lambda(b)) \leq \lambda(r)$

(II) The simplification $(a \Rightarrow (b \Rightarrow c)) \Leftrightarrow a\wedge b \Rightarrow c$ yields the first four clauses of the conclusion by simple multiplication and use of the transitivity of $\leq$. It remains to show $y+1\leq\lambda(y)$ in the conclusion.The equality in the premise yields:

$$y = a\text{-}x/b \leq a\text{-}b/b = a/b - 1 \text{ , hence}$$
$$y+1 \leq a/b \leq a \leq \lambda(a)$$

Therefore we have

(1) $y+1 \leq \lambda(a) \Rightarrow y+1 \leq \lambda(y)$, and with (*):

(iii) $\lambda(a) \leq \lambda(y)$

This condition is explained by the fact, that the maximal value y can obtain, is the value of a (i.e. in the case that $b=1$).

(III) Arithmetical simplifications like $0b=0$ or $0+a=a$ and ordering of the clauses yields the first three and the two last clauses of the conclusion. Then we apply (*) and consider $\lambda(x) \geq 0$ as an axiom so that the following condition remains

(iv) $\lambda(a) \leq \lambda(x)$,
because the maximal value, which x can obtain, is the value of a.

Acknowledgement:
This work was to one part supported by the "Sonderforschungsbereich 314 Künstliche Intelligenz" and to the other by a special grant from IBM Germany, which we gratefully acknowledge.
Special thanks go to Walter Olthoff and Jörg Siekmann, who read an earlier version of the paper and contributed useful hints.

Literature:

[AA 78] Alagic,S.,Arbib,M.A.: The Design of Well-Structured and Correct Programs. Springer, New York, 1978.

[ANSI 74] American National Standard Institute: Programming Language COBOL, ANSI X3.23-1974, New York 1974.

[Ba 80] deBakker,J.W.: Mathematical Theory of Program Correctness, Prentice-Hall, London, 1980.

[BM 80] Boyer,R.S., Moore,J.S.: A Verification Condition Generator for Fortran, Technical Report CSL-103 SRI Project 4079, Menlo Park, 1980.

[Flo 74] Ch. Floyd: Strukturierte Programmierung für COBOL Anwender, Hoffmann und Campe, Hamburg 1974.

[Hoa 71] Hoare,C.A.R.: Procedures and Parameters: An axiomatic Approach. In: Engeler,E. (ed.): Symposium on Semantics of Algorithmic Languages. Lecture Notes in Mathematics 188. Berlin - Heidelberg - New York, Springer 1971, 102-116.

[HW 73] Hoare,C.A.R., Wirth,N.: An Axiomatic Definition of the Programming Language PASCAL, Acta Informatica 2 (1973), 335-355.

[ILL 73] Igarashi,S.,London,R.,Luckham,D.:Automated Program Verification I. A Logical Basis and its Implementation. Acta Informatica 4, 145-182(75).

[Ma 75] Marmier,E.: Automatic Verification of Pascal Programs. Dissertation at the Swiss Federal Institute of Technology, Zürich, Zürich 1975.

[Ol 83] Olderog,E.-R.: Note on the Notion of Expressiveness and the Rule of Adaption. Theor. Comp. Science 24 (1983), 337-347.