Applied Mathematics & Information Sciences **1**(2)(2007), 195-202
— An International Journal
©2007 Dixie W Publishing Corporation, U. S. A.

A Partial Correctness Proof for Programs with Decided Specifications

A. A. Darwish
Mathematics Department, Faculty of Science, Helwan University, Cairo, Egypt
Email Address: amodarwish@yahoo.com

Received 2 September 2006; Accepted 20 January 2007

This paper provides the method and complete proof for programs written in Pascal programming language with decided specifications for programs which reverse the digits of an integer from [5]. The author of this paper describes a new concept of partial correctness of programs better suited to specification purposes than the classical one. Partial correctness specifications are pairs of assertions, preconditions and postconditions. As an application of partial correctness specifications, the paper presents the correctness method for some of the programs which have been written in procedural programming language. Moreover, this method is suitable for all procedural programs.

Keywords: Partial correctness, axiomatic semantics, loop invariant, precondition, postcondition.

1 Introduction

It was Hoare's paper [9] that introduced the now well known notation for partial correctness of programs on the form $P\{Q\}R$, where P and R are predicates specifying the preconditions and postconditions (desired result), respectively, for program Q. That is, if the assertion P is true before the initiation of a program Q, then assertion R will be true when it completes execution. This paper is based on an earlier work of Floyd [8] where the technique was applied to flowcharts rather than to programming language text. Hoare's axiomatic semantics was also fundamental to the development of sequential systems [1] and for the real time specification language, called ASTRAL [4]. From the practical point of view, it is suitable to combine loop invariants with termination constraints to keep the distinction between partial and total correctness. [2] gives an operational and axiomatic framework for both partial and total correctness of probabilistic and demonic sequential programs; among other aspects, that provide the theory to support the practical publications on probabilistic demonic loops [3]. Assembler programs contain unstructured jumps and previous formalisms have modeled these by using continuations. [12] provides an approach, which uses techniques from the compiler theory.

This paper begins with introducing the notion of assertion proofs for the program correctness. Then, it introduces some examples of annotation of programs. As an illustration, an application is made to one of the programs written in Pascal language; therefore, the following sections are concerned with proving the partial correctness of programs.

2 Some Elementary Examples of Assertion Proofs about Program Correctness

2.1 Verification of complex programs versus simple programs

In programming of various problems, it is possible to allocate some consecutive stages in which the process of programming can be divided into the following: 1) problem formalization; 2) refinement (specification) of formalized problem; 3) construction of programs; 4) debugging; 5) program improvement. The last item can be repeated many times. The correct annotating of programs presents the original theoretical program debugging.

To annotate programs [8, 9], the help of approval of programs, which will be in various locations of the program (encompassed by curly bracket), is suggested. The program annotation is correct if before the execution the next step and after the assertion operator is supposed, the program can execute this assertion. Following are some simple examples of the annotation of programs:

$$(x := 0\{x = 0\}), (< x = 0 > \; x := x + 2\,; \{x = 2\}),$$

condition $x = 0$ is precondition and $x = 2$ is postcondition for the last assertion.

The annotation for more complicated programs is available from annotation for simple programs. For example, from issued higher annotated programs obtain

$$(x := \{x = 0\}, x := x + 2\,; \{x = 2\}).$$

The program is considered to be correctly annotated if it is correctly annotated every time during the execution, since the step of the program is directly carried out after the assertion of any comment, can be considered as condition, and is supposed before the next step of the program.

We will consider that the correct annotation $(P; (AQ))$ is equivalent to the correct annotation of $(P\{A\})$ and (AQ). Here and in the following P, Q, and R are fragments of a program while A, B, and C are verifications (annotations).

Another example, from

$$(< x = 2 \& x \geq 0 > y := 0; \{y = 0\}), (< x = 2 \& \neg(x \geq 0) > \{y = 0\})$$

(the last expression is equivalent to the verification $x = 2\&\neg(x \geq 0) \supset y = 0)$ to get

$$(< x = 2 > \text{ IF } x \geq 0 \text{ THEN } y := 0;\ \text{FI}\{y = 0\}).$$

Let us write the used method in the following generalized form:
For correct annotation

$$(< B\&A > PQ), (< B\&\neg A > Q).$$

Equivalence of correct annotation

$$(< B > \text{ IF } A \text{ THEN } P \text{ FI};\ Q).$$

An additional example: from

$$< T\&x \geq 0 > y := x;\ \{y = |x|\},\ (< T\&\neg(x \geq 0) > \ y := -x; \{y = |x|\}),$$

to get

$$(< T > \text{ IF } x \geq 0 \text{ THEN } y := x;\ \ \text{ELSE } y := -x; FI\{y = |x|\}).$$

Here T refers to the logical constant: true. The same method in the generalization form:
Correct annotation programs

$$(< A\&B > QP), (< A\&\neg B > R;\ P).$$

It is equivalent to a correctness of the annotation of the program

$$(< A > \text{ IF } B \text{ THEN } Q \text{ ELSE } R \text{ FI};\ P).$$

Finally, consider a much more complex example: from

$$(< x \geq 0\&x = v\&y = 1\& \neq (x \geq 1) > \{y = 2^{[\nu]}\&[x] = 0\})$$

(it is not executing any step of the loop),

$$(< 0 \leq x\&x = v\&y = 1\&x \geq 1 > \ x := x - 1;\ y := y * 2; \{x \geq 0\&y = 2^{[\nu]-[x]}\})$$

(execute the first step of the loop),

$$(< x \geq 1\&x \geq 0\&y = 2^{[\nu]-[x]} > x := x - 1;\ y := y * 2; \{x \geq 0\&y = 2^{[\nu]-[x]}\})$$

(multiple execution of the loop),

$$(< \neg(x \geq 1)\&x \geq 0\&y = 2^{[\nu]-[x]} > \{y = 2^{[\nu]}\&[x] = 0\})$$

(output of the loop) is received

$$(< x \geq 0 \ \& x = v \& y = 1 > \text{ WHILE } x \geq 1, \text{ DO}$$
$$x := x - 1; \ y := y * 2; \{x \geq 0 \& y = 2^{[\nu]-[x]}\} \text{ OD}; \{y = 2^{[\nu]} \& [x] = 0\}).$$

The same rule in the generalization form: a correctness of the annotation of programs [11].

$$(< A \& \neg B > Q), \ (< A \& B > P\{C\}),$$
$$(< C \& B > P\{C\}), \ (< C \& \neg B > Q),$$

which equivalent to the correct annotation of the program

$$(< A > \text{ WHILE } B \text{ DO } P\{C\} \text{ OD}; \ Q).$$

2.2 Loop invariant

Discovering the loop invariant requires insight. Consider the following program fragment that calculates factorial, as indicated by the final assertion.

```
$(< N \geq 0 >$
$k := N; \ f := 1;$
While $k > 0$ do
$f := f * k; \ k := k - 1; \ \{\text{loop invariant}\}$
end while
$\{f = N!\})$
```

The loop invariant involves a relationship between variables that remains the same regardless of how many times the loop executes. The loop invariant also involves the while loop condition $k > 0$ in the example above, modified to include the exit case, which is $k = 0$ in this case. Combining these conditions, we have $k \geq 0$ as part of the loop invariant. Now we have our loop invariant: $\{f * k! = N! \text{ and } k \geq 0\}$

Some principles to construct loop invariant

Constructing loop invariants in programs provides the main challenge when proving correctness. Several general principles can assist in analyzing the logic of the loop when finding an invariant [10].

- A loop invariant describes a relationship among the variables that does not change as the loop is executed. The variables may change their values, but the relationship stays constant.

- Constructing a table of values for the variables that change often reveals a property among variables that does not change.

- Combining what has already been computed at some stage in the loop with what has yet to be computed may yield a constant of some type.

- An expression related to the test A for the loop can usually be combined with the assertion {not A} to produce part of the postcondition.

- A possible loop invariant can be assembled to carry out the proof.

In this paper, the author provides proof of the correctness of program which can reverse the digits of an integer from [5]. To perform this task, the proof is divided into many steps and each time we will prove the partial correctness of the step as a separate program to obtain a partially correct program. The proof depends on Hoare's axiomatic semantics with the loop invariant [9].

3 Program: Reversing the Digits of an Integer

Problem

Design an algorithm that accepts a positive integer and reverses the order of its digits.

Algorithm description

1- Establish n, the positive integer to be reversed.

2- Set the initial condition for the reversed integer dreverse.

3- While the integer being reversed is greater than zero do (a) use the remainder function to extract the rightmost digit of the number being reversed; (b) increase the previous reversed integer by a factor of 10 and add to it the most recently extracted digit to give the current drivers value; (c) use integer division by 10 to remove the rightmost digit from the number being reversed.

This algorithm is most suitably implemented as a function which accepts the integer to be reversed as input and returns the integer with its digits reversed as output.

Pascal Implementation

function *dreverse* (n: **integer**): **integer**;

var *reverse*: **integer**;

$$\left\{ n \succ 0 \& n = \left[\sum_{i=0}^{64} (10^i * a[i]) \right] \& \forall i (0 \leq i \leq 64 \Rightarrow a[i] \geq 0) > \right\}$$

```
begin {reverse the order of the digits of a positive integer}
```

$$\left\{ \begin{array}{c} n = \left[\sum_{i=0}^{64}(10^i * a[i]\right] \& \forall_i (0 \le i \le 64 \Rightarrow a[i] \ge 0) \& \exists k' (0 \le k' \le 64 \\ \& a[k'] \succ 0 \& \forall_j (j \succ k' \Rightarrow a[j] = 0)) \end{array} \right\}$$

reverse: =0;

$$\left\{ \begin{array}{l} \textbf{invariant:} \\ \exists k' (n = \left[\sum_{i=0}^{k'-j}(10^i * a[i]\right] \& n \ge 0 \& \forall_i (0 \le i \le 64 \Rightarrow a[i] \ge 0) \& 0 \le k' \le 64 \& \\ \quad reverse = \left[\sum_{i=0}^{j-1}(10^i * a[k-i]\right] \& \forall_j (j > k' \Rightarrow a[j] = 0) \& a[k'] > 0) \end{array} \right\}$$

```
while $n \succ 0$ do
        begin
        reverse: = reverse* 10 + $n$ mod 10;
        $n$: = $n$ div 10{invariant}
        end;
```

$$\left\{ \begin{array}{c} \textbf{conclusion: } \exists k' \; (0 \le k' \le 64 \; \& \; reverse = \left[\sum_{i=0}^{k'}(10^i * a[k'-i]\right] \; \& \\ \forall_j ((j > k') \Rightarrow (a[j] = 0)) \; \& \; a[k'] > 0) \end{array} \right\}$$

```
dreverse: = reverse
end
```

To describe the complete proof of correctness for this program, we need to divide this program into many subprograms (fragments and then prove the correctness of each one as a separate program). So we have preconditions and postconditions as follows:

Case 1

Precondition:

$$\left\{ n \succ 0 \& n = \left[\sum_{i=0}^{64}(10^i * a[i])\right] \& \forall i (0 \le i \le 64 \Rightarrow a[i] \ge 0) > \right\}$$

Postcondition:

$$\left\{ \begin{array}{c} n = \left[\sum_{i=0}^{64}(10^i * a[i]\right] \& \forall_i (0 \le i \le 64 \Rightarrow a[i] \ge 0) \& \exists k' (0 \le k' \le 64 \\ \& a[k'] \succ 0 \& \forall_j (j \succ k' \Rightarrow a[j] = 0)) \end{array} \right\}$$

Statement: empty statement

Case 2

Precondition:

$$\left\{\begin{array}{c} n = \left[\sum_{i=0}^{64}(10^i * a[i]\right] \& \forall_i (0 \le i \le 64 \Rightarrow a[i] \ge 0) \& \exists k' (0 \le k' \le 64 \\ \& a[k'] \succ 0 \& \forall_j (j \succ k' \Rightarrow a[j] = 0)) \end{array}\right\}$$

Statement: reverse $:= 0;$

Postcondition:

$$\left\{\begin{array}{l} \textbf{invariant:} \\ \exists k'(n = \left[\sum_{i=0}^{k'-j}(10^i * a[i]\right] \& n \ge 0 \& \forall_i (0 \le i \le 64 \Rightarrow a[i] \ge 0) \& 0 \le k' \le 64 \& \\ \quad reverse = \left[\sum_{i=0}^{j-1}(10^i * a[k-i]\right] \& \forall_j (j > k' \Rightarrow a[j] = 0) \& a[k'] > 0) \end{array}\right\}$$

Case 3

Precondition:

$$\left\{\begin{array}{l} \textbf{assert:} \\ \exists k'(n = \left[\sum_{i=0}^{k'-j}(10^i * a[i]\right] \& n \ge 0 \& \forall_i (0 \le i \le 64 \Rightarrow a[i] \ge 0) \& 0 \le k' \le 64 \& \\ reverse = \left[\sum_{i=0}^{j-1}(10^i * a[k-i]\right] \& \forall_j (j > k' \Rightarrow a[j] = 0) \& a[k'] > 0) \& n \succ 0 \end{array}\right\}$$

Program: $reverse := reverse * 10 + n$**mod** $10;\ n := n$ **div** 10

Postcondition:

$$\{invariant\}$$

Case 4

Precondition:

$$\{invariant\ \& \neg\ (n \succ 0)\}$$

Postcondition:

$$\left\{\begin{array}{c} \textbf{conclusion: } \exists k'\ (0 \le k' \le 64\ \&\ reverse = \left[\sum_{i=0}^{k'}(10^i * a[k'-i]\right] \& \\ \forall_j ((j > k') \Rightarrow (a[j] = 0))\ \&\ a[k'] > 0) \end{array}\right\}$$

All cases are correct programs with preconditions and postcondition. So the main program is correct. This example refines the example from [3]. One way to relate partial and total correctness is by the informal equation partial correctness + termination = total correctness.

For more details of such specification, see [6, 7].

References

[1] B. Auernheimer and R. A. Kemmerer, RT-ASLAN: A specification language for real-time systems, *IEEE Transactions on Software Engineering*, **12**(1986), 879-889.

[2] A. K. McIver and C. Morgan, Partial correctness for probabilistic demonic programs, *Theoretical Computer Science*, **266**(2001), 513-541.

[3] C. C. Morgan, *Proof rules for probabilistic loops*, in: Proc. BCS-FACS Seventh Refinement Workshop, Workshops in Computing, Springer, Berlin, 1996.

[4] A. Coen-Porisini, C. Ghezzi, and R. A. Kemmerer, Specification of realtime systems using ASTRAL, *IEEE Transactions on Software Engineering*, **23**(1997), 572-598.

[5] R. G. Dromey, *How To Solve It By Computer*, Prentice Hall International Series in Computer Science, London, 1982.

[6] E. C. R. Hehner, *A Partial Theory of Programming*, Springer, 1993.

[7] E. C. R. Hehner, *Specifications, Programs, and Total Correctness*, Science of Computer Programming (Sci. comput. program.), **34**, No. 3, Elsevier, 1999.

[8] R. W. Floyd, Mathematical aspects of computer science, *Proceedings of Symposia in Applied Mathematics*, 19-32, 1967.

[9] C. A. R. Hoare, An axiomatic basis for computer programming, *Communications of the ACM* **12**(1967), 576-583.

[10] K. Slonneger and B. L. Kurtz, *Formal Syntax and Semantics of Programming Languages*, Addison-Wesley Publishing Company, USA, 1995.

[11] N. K. Kossovski, *Elements of Mathematical Logic and its Application to the Theory of Sub-recursive Algorithms*, Saint Petersburg State University, Saint Petersburg, 1981 (in Russian).

[12] G. Watson and C. Fidge, A partial-correctness semantics for modeling assembler programs, *Software Engineering and Formal Methods Proceeding*, 82-90, IEEE, 2003.