An implementation of mutual inclusion

Rob Hoogerwoord
Department of Mathematics and Computing Science
Eindhoven University of Technology
5600 MB Eindhoven
The Netherlands

Abstract

We consider the parallel composition of two cyclic programs. The interaction of these programs consists of a form of synchronisation sometimes referred to as "mutual inclusion". For a given implementation of this synchronisation by means of semaphore operations we prove the correctness of the programs and we prove the absence of the danger of deadlock.

0. Introduction

We consider the parallel composition of two programs. The interaction of these programs consists of a form of synchronisation sometimes referred to as "mutual inclusion", indicating that some parts of the programs have to be executed "more or less simultaneously". We shall not try to define what is meant by "more or less simultaneously". Instead, we investigate the properties of two very specific, cyclic programs that may be considered as prototype programs with respect to mutual inclusion. The programs are:

```
A:  a := 0                        B:  b := 0
  ; do true → clicka                ; do true → clickb
             { a = b }                         { b = a }
             ; clicka                          ; clickb
             ; a := a + 1                      ; b := b + 1
    od                                od
```

The operations "clicka" and "clickb" both are instances of the general operation "click"; all clicks in program A have been suffixed with "a" and all clicks in program B have been suffixed with "b". Thus we are enabled to consider implementations of click that are not necessarily identical for each of the two programs. The operational interpretation of the click operations is that any such operation in one of the programs is eligible for execution only if one of the click operations in the other program is also eligible for execution; in that case both clicks are equivalent to "skip".

W.H.J. Feijen suggested the following implementation of the click operations, using two semaphores x and y , the initial values of which are both zero:

```
clicka:   V(x) ; P(y)
clickb:   V(y) ; P(x)
```

Following this suggestion we prove the correctness of this implementation. Herewith, we postulate that we call any proposed implementation of click correct if with that implementation the two prototype programs A and B are correct in the following sense:

- the occurrence of the assertions $a = b$ and $b = a$ in the above programs is justified, and:
- the programs are free from the danger of deadlock.

The decision whether or not this postulate captures the quintessence of mutual inclusion is left to the reader; our subject is the development of the following proofs. The first step of this development will be the elimination of the semaphore operations by considering them as "special" operations on integer variables.

1. Elimination of the semaphore operations

One possible approach to prove properties of programs containing semaphore operations is to define the semaphore operations as "special" operations on integer variables and, then, to apply the Gries-Owicki theory [0,1] to the programs thus obtained. Application of the Gries-Owicki theory implies the formulation of a set of predicates and invariant relations by means of which properties of the programs can be proved. In our case, we wish to prove that $a = b$ holds at a certain place in program A and, symmetrically, that $b = a$ holds at a certain place in program B.

(Note: On account of the symmetry of the programs the proof obligation is symmetric too; hence, it suffices to prove one half of it. In the sequel we shall exploit this symmetry as much as possible without saying so every time).

We observe that a and b are local variables and that the interaction of A and B consists in semaphore operations only. So, we most certainly will need a relation between variable a and semaphores x and y , and a relation between b and y and x . These two relations shall be such that the equality of a and b can be derived from them. It seems, however, difficult to find such relations because a and b can assume arbitrarily large values whereas x and y never exceed 2. Therefore, we take the following approach.

Each semaphore s is represented by a pair ps,vs of integer variables satisfying $s = vs - ps$. Each operation P(s) is coded as $ps := ps + 1$ and each operation V(s) is coded as $vs := vs + 1$. The property that s assumes only natural values is reflected by the relation $ps \leq vs$, which we shall call the "semaphore invariant". In the sequel we consider $ps := ps + 1$ and $vs := vs + 1$ as atomic actions and we assume that the mechanism executing the programs interleaves the atomic actions of the programs in such a way that the semaphore invariants of all semaphores in the programs are universally true. Application of this transformation and addition of some assertions yields the following programs:

```
A:  a,vx,py := 0,0,0 { P0 }
  ; do true → { P0 } vx := vx + 1 { P1 }
                   ; py := py + 1 { P2 }
                     { a = b }
                   ; vx := vx + 1 { P3 }
                   ; py := py + 1 { P4 }
                   ;  a :=  a + 1 { P0 }
    od
```

B: obtained from A by interchanging all a and b , x and y , and P and Q .

For P0 we make a choice; the predicates P4,P3,P2,P1 are derived from P0 by repeated application of the axiom of assigment:

P0:	$2 * a = vx$	$\wedge$	$2 * a = py$
P4:	$2 * a + 2 = vx$	$\wedge$	$2 * a + 2 = py$
P3:	$2 * a + 2 = vx$	$\wedge$	$2 * a + 1 = py$
P2:	$2 * a + 1 = vx$	$\wedge$	$2 * a + 1 = py$
P1:	$2 * a + 1 = vx$	$\wedge$	$2 * a = py$

Because { P0 } vx := vx + 1 { P1 } holds as well we conclude that P0 indeed is an invariant of program A's repetition.

2. Proof of correctness

We start this section by noting that, as a result of the transformation, variables a, vx, and py are local variables of program A. Hence, P0 through P4 are trivially invariants of program B. Furthermore, each of the predicates Pi $(0 \le i < 5)$ satisfies Pi => P , where:

P: $vx - 2 \le 2 * a \le py$

Hence, P is a global invariant of both programs. Similarly, by exploitation of the symmetry, we find that Q is a global invariant of both programs, where:

Q: $vy - 2 \le 2 * b \le px$

Finally, the interaction of the two programs is expressed by the conjunction S of the semaphore invariants:

S: $px \le vx \;\wedge\; py \le vy$

<u>Lemma0</u>: $(P2 \wedge Q \wedge S) \Rightarrow (a = b)$
<u>proof</u>: Assuming that $P2 \wedge Q \wedge S$ holds, we derive:

$$\begin{aligned} 2 * a &= \{ P2 \}\; py - 1 \\ &\le \{ S \}\; vy - 1 \\ &\le \{ Q \}\; 2 * b + 1 \end{aligned}$$

Hence, because a and b are integers: $a \le b$.
Similarly, we derive:

$$\begin{aligned} 2 * b &\le \{ Q \}\; px \\ &\le \{ S \}\; vx \\ &= \{ P2 \}\; 2 * a + 1 \text{ ; hence: } b \le a \text{ .} \end{aligned}$$

Combination of the two results gives: $a = b$.
(End of proof).

3. <u>Absence of the danger of deadlock</u>

The semaphore invariant restricts the freedom of the implementation to select a "next" atomic action: an atomic action may only be selected if it does not violate the semaphore invariant. If, due to this restriction, in a given state no atomic action can be selected such a state is called a deadlock state and the computation is said to suffer from deadlock. Proving the absence of the danger of deadlock then is proving that deadlock states do not occur. In program A the only atomic action that could violate S is the P operation $py := py + 1$, namely when $py = vy$. Because $P1 \vee P3$ is a precondition of any operation $py := py + 1$ in program A, any deadlock state satisfies: $(P1 \vee P3) \wedge py = vy$.

Similarly, by symmetry, deadlock states satisfy: $(Q1 \vee Q3) \wedge px = vx$. Finally, all states satisfy S and so do deadlock states. Hence, any deadlock state satisfies D , where:

D: $(P1 \vee P3) \wedge (Q1 \vee Q3) \wedge py = vy \wedge px = vx \wedge S$

Lemma1: $\neg D$

proof:

```
D  =  { definition of D and S, and calculus }
      $(P1 \vee P3) \wedge (Q1 \vee Q3) \wedge py = vy \wedge px = vx$
   => { both P1 and P3 imply  $vx = py + 1$ }
      $vx = py + 1 \wedge (Q1 \vee Q3) \wedge py = vy \wedge px = vx$
   => { both Q1 and Q3 imply  $vy = px + 1$ }
      $vx = py + 1 \wedge vy = px + 1 \wedge py = vy \wedge px = vx$
   =  { calculus }
      $px = py + 1 \wedge py = px + 1 \wedge py = vy \wedge px = vx$
   =  { calculus }
      false.
```

(End of proof).

From lemma1 we conclude that deadlock states do not occur. In the more general case of two programs containing m and n P operations respectively, m times n pairs of P operations exist that correspond to possible deadlock states; so, a proof of the absence of deadlock implies verification of m times n cases. In our example the 4 cases, as represented by the 2 disjunctions in the formula $(P1 \vee P3) \wedge (Q1 \vee Q3)$, have enough in common to coincide.

4. Epilogue

The decision to divide the semaphores x and y into the pairs px,vx and py,vy respectively was inspired by the observation that $2 * a =$ "the number of completed P(y) operations" is an invariant of program A's repetition. A pleasant consequence of this decision is the fact that after the transformation the two programs contain local variables only, but honesty forces us to admit that this property was not a priori intended. Another consequence of the transformation is that all synchronisation of the programs is captured by -- one might also say: "is hidden behind" -- the semaphore invariant, the invariance of which is taken for granted. From our point of view, viz. the desire to prove properties of the programs, this property is rather pleasant as it opens the way to a remarkably simple proof. Finally, we note that the semaphore invariant is the only knowledge about semaphores we have used, which corresponds to the weakest possible interpretation of semaphores. As a consequence, the argument given in this paper is independent of any definition of semaphores one may have in mind, as long as the semaphore invariant is satisfied.

5. Acknowledgements

Thanks are due to W.H.J. Feijen, for posing the problem, and to the members of the Eindhoven Tuesday Afternoon Club, for their suggestions and criticism with respect to the presentation.

References

[0] S. Owicki and D. Gries: "An axiomatic proof technique for parallel programs I", Acta Informatica 6, 319-340 (June 1976).

[1] Edsger W. Dijkstra: "Selected writings on computing: a personal perspective", chapter: "A personal summary of the Gries-Owicki theory". (Springer-Verlag New York Inc.,1982).

(Eindhoven, 1985.8.19)