Parallel Usage Checking – An Observation

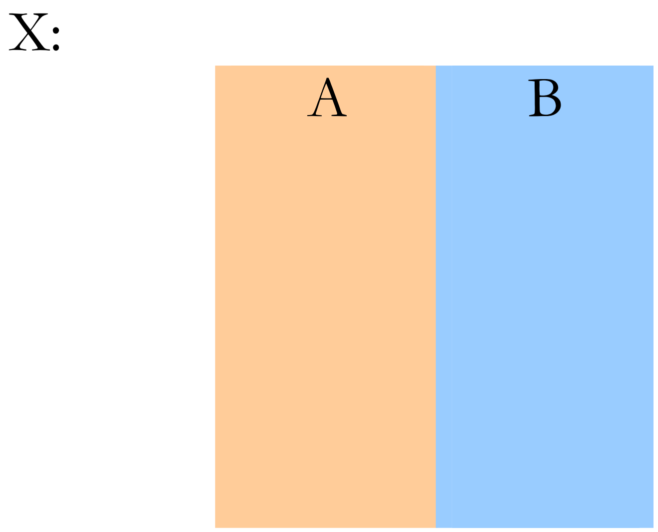
Barry M Cook

4Links Limited, UK

SpaceWire Designers, Consultants, Manufacturers

Parallel Usage Checking …

```
X: -- a “shared” variable
PAR
  A – a process that ‘uses’ X
  B – a process that ‘uses’ X
```



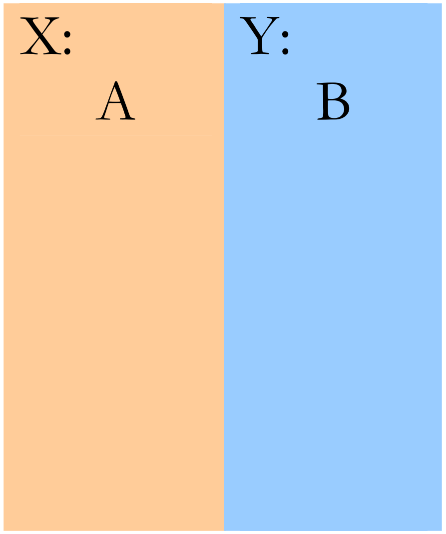
… detects a possible problem if A or B or both write to X

Require: **CREW – Concurrent Read, Exclusive Write**

An acceptable program:

Give each parallel process its own variable and send data between processes through a communication channel

C: -- a communication channel



X is local to A and Y is local to B and there is no conflict

A typical use:

Process A (say) places data in its variable, X, and – at some point – passes these values to process B where they are stored in its local variable, Y, for use.

C: -- a communication channel

X:	Y:
SEQ	SEQ
…	…
-- fill X	C ? Y
…	…
C ! X	-- use Y
…	…

Note: the relative time of action is NOT accurately represented by the relative positions of words in the above picture …
In fact, the communication *synchronises* the processes

Re-drawing the picture:

C: -- a communication channel

X:	Y:
SEQ	SEQ
A_1	B_1
C ! X	C ? Y
A_2	B_2

The synchronising communication divides each process into temporally distinct parts – and we can see that it is perfectly safe for A_1 and B_2 (or B_1 and A_2) to use a shared variable:

C: -- a synchronising channel (no data)
X: -- a shared variable

SEQ	SEQ
A_1 -- uses X	B_1
C !	C ?
A_2	B_2 -- uses X

Efficiency

Using separate variables and communicating (possibly a large amount of) data can be slow.

Using a shared variable requires no data transfer and can be much more efficient.

Formalising the efficient version

We can re-write the shared variable version:

C: -- a synchronising channel (no data)
X: -- a shared variable

SEQ	SEQ
A_1	B_1
C !	C ?
A_2	B_2

As:

C: -- a synchronising channel (no data)
X: -- a shared variable
SEQ

A_1	B_1

-- communicate

A_2	B_2

And parallel-usage check it in the usual way.

Or

```
X: -- variable
C: -- channel
PAR
  SEQ
    A1
    C !
    A2
  SEQ
    B1
    C ?
    B2
```

is equivalent to

```
X: -- variable
C: -- channel
SEQ
  PAR
    A1
    B1
  -- communicate
  PAR
    A2
    B2
```

Represents the required solution

Proves that it is safe
- but is probably not a good implementation

But …

We do need to make sure the behaviour is adequately controlled and the transformation is valid.

e.g. a loop in the above example …

```
WHILE TRUE
  SEQ
    A1
    C !
    A2
```

… means that A_1 is both before and after A_2

(a solution is to use another communication to synchronise after A_2/B_2)

Why not just write the transformed version?

1.

It may be less efficient (see above)

2.

"There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies and the other is to make it so complicated that there are no obvious deficiencies."

Professor Sir C.A.R "Tony" Hoare

It is better to write programs in a way that reflects the problem solution – and is easily seen to be correct.

Conclusions

Parallel usage checking is required.

Program transformation before checking can allow a larger range of acceptable programs
… that may be more efficiently implemented

We often think of program transformations as steps towards implementation

I suggest that we might also use (possibly different) transformations purely / additionally as steps towards correctness checking