

Gate-level modelling and verification of asynchronous circuits using CSP_M and FDR

Mark B. Josephs

Centre for Concurrent Systems and VLSI
London South Bank University

12 March 2007

CSP stands for Communicating Sequential Processes.

It was originally introduced as a parallel programming language by Tony Hoare in his 1978 article in Communications of the ACM.

His paper influenced the development of occam, a language for the design and programming of concurrent systems.

Groups at Caltech and Philips Research now have twenty-years' experience of syntax-directed translation from languages based on CSP into asynchronous VLSI circuits.

This talk has nothing to do with the 1978 paper, nor syntax-directed translation!

In the 1980s, foundational research by Hoare's group at Oxford focused on a process algebra (sometimes referred to as Theoretical CSP) in which the imperative aspects of the original language were suppressed.

A number of semantic models for (T)CSP were developed, the *failures/divergences* model from 1985 being the standard one.

The notion of *refinement* between a specification and an implementation was also formalised.

The 1990s saw the development of a model-checker, FDR, for verifying properties of processes.

FDR analyses processes that are expressed in a machine-readable dialect of CSP (known as CSP_M) that includes a functional-programming language.

See <http://www.fsel.com> and
<http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/68b.pdf>

The output signal of a digital logic gate is either *stable* or *excited*.

Stability can be captured by a Boolean equation or inequation,

e.g., $C == (A \text{ and } B)$ for an AND-gate;

e.g., $C != (A \text{ and } B)$ for a NAND-gate.

It can be formalised in CSP_M as a function from a sequence of Booleans (the state vector) to a Boolean, where the Boolean value of the output signal is included as the first element in the sequence.

$$\text{inv_eqn}(\langle B, A \rangle) = B != A$$

$$\text{and_eqn}(\langle C, A, B \rangle) = C == (A \text{ and } B)$$

$$\text{c_eqn}(\langle C, A, B \rangle) =$$

$$C == ((A \text{ and } B) \text{ or } (B \text{ and } C) \text{ or } (C \text{ and } A))$$

In CSP_M ,

$XS^{\wedge}YS$ denotes the catenation of sequences XS and YS ;

$elem(X, XS)$ tests whether an element X occurs in a sequence XS .

Thus our functions can be generalized to $\#XS$ -input gates:

$$and_eqn(\langle Y \rangle^{\wedge} XS) = Y \neq elem(false, XS)$$

$$nand_eqn(\langle Y \rangle^{\wedge} XS) = Y == elem(false, XS)$$

$$or_eqn(\langle Y \rangle^{\wedge} XS) = Y == elem(true, XS)$$

$$c_eqn(\langle Y \rangle^{\wedge} XS) = \\ Y == ((not\ elem(false, XS))\ or\ (Y\ and\ elem(true, XS)))$$

The switching behaviour of a gate can be captured in a few rules:

If its output signal is stable, then only an input signal may switch.

If its output signal is excited, then either that signal or an input signal may switch.

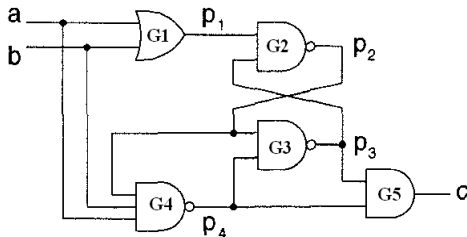
Computation interference occurs when the switching of an input signal causes the output signal to change from excited to stable.

```
BOTTOM = BOTTOM |~| BOTTOM
```

```
complement(i, <X>^XS)  
= if i == 0  
  then <not X>^XS  
  else <X>^complement(i-1, XS)
```

```
channel ins : Int  
channel out
```

```
GATE(eqn)(Y,XS)
= let G = GATE(eqn)
    S = 0..#XS-1
  within
  if eqn(<Y>^XS)
  then ins?i:S -> G(Y,complement(i,XS))
  else ins?i:S -> ( let XS' = complement(i,XS)
                    within
                    if eqn(<Y>^XS')
                    then BOTTOM
                    else G(Y,XS') )
  [] out -> G(not Y,XS)
```



channel a, b, c, p1, p2, p3, p4

G1 = GATE(or_eqn)(false,<false,false>) [[out<-p1, ins.0<-a, ins.1<-b]]

G2 = GATE(nand_eqn)(true,<false,false>) [[out<-p2, ins.0<-p1, ins.1<-p3]]

G3 = GATE(nand_eqn)(false,<true,true>) [[out<-p3, ins.0<-p2, ins.1<-p4]]

G4 = GATE(nand_eqn)(true,<false,false,true>) [[out<-p4, ins.0<-a, ins.1<-b, ins.2<-p2]]

G5 = GATE(and_eqn)(false,<false,true>) [[out<-c, ins.0<-p3, ins.1<-p4]]

IMP

= (((G1 | [{a,b}] | G4) | [{p1,p2,p4}] | (G2 | [{p2,p3}] | G3)) | [{p3,p4}] | G5) \ {p1, p2, p3, p4}

Most instances of GATE are output-delay-insensitive.

An exception is when both $\text{eqn}(\langle \text{true} \rangle^X S)$ and $\text{eqn}(\langle \text{false} \rangle^X S)$ are false.

If either state were reachable, the gate would repeatedly switch between them — something quite useless in this setting!

Output-delay-insensitisation would transform this into divergence.

When a signal is forked to several gates, our CSP_M model of the circuit treats each end of the fork as switching value at the same time.

This can be relaxed by applying input-delay-insensitisation (with respect to that signal) to any or all of the gates to which it forks, i.e., asymmetric isochronic forks and non-isochronic forks can be readily modelled.

The circuit we considered above is known as Mayevsky's implementation of a C-element. Unfortunately,
`assert not C[[ins.0<-a, ins.1<-b, out<-c]] [FD= IMP`

It does, however, implement a Join element:

```
JOIN = IDI({a,b}, C[[ins.0<-a, ins.1<-b, out<-c]])  
assert JOIN [FD= IMP
```

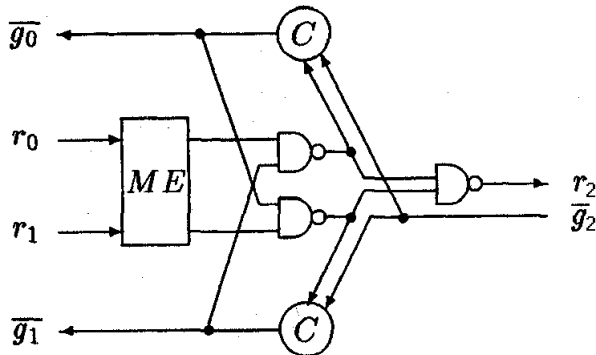
It also turns out that three of its isochronic forks can be made asymmetric:

```
assert JOIN [FD=  
(((G1 | [a,b] | IDI({p2},G4)) | [{p1,p2,p4}] |  
  (G2 | [{p2,p3}] | IDI({p4},G3)))  
  | [{p3,p4}] | IDI({p3},G5)) \ {p1, p2, p3, p4}
```

```
mutex_eqn(<Y,X>^YS) =  
  Y == ( X and (not elem(true,YS)) )  
  
MUTEX(<Y0,Y1,X0,X1>  
=   ins.0 -> ( if mutex_eqn(<Y0,X0,Y1>  
                then MUTEX(<Y0,Y1,not X0,X1>  
                else BOTTOM )  
  [] not mutex_eqn(<Y0,X0,Y1> ) &  
    outs.0 -> MUTEX(<not Y0,Y1,X0,X1>)  
  [] ins.1 ...  
  [] not ...  
  
ME =  
  ODI({outs.0,outs.1}, MUTEX(<false,false>,<false,false>))
```

```
channel r, gbar : Int
```

```
TA =    r.0 -> r.2  
        -> gbar.2 -> gbar.0  
        -> r.0 -> r.2  
        -> gbar.2 -> gbar.0 -> TA  
[] r.1 -> r.2  
    -> gbar.2 -> gbar.1  
    -> r.1 -> r.2  
    -> gbar.2 -> gbar.1 -> TA
```



```

ME' = ME[[ins.0<-r.0, ins.1<-r.1, outs.0<-mids.0, outs.1<-mids.1]]
G0 = GATE(nand.eqn)(true,<false,true>) [[out<-mids.2, ins.0<-mids.0, ins.1<-gbar.1]]
G1 = GATE(nand.eqn)(true,<false,true>) [[out<-mids.3, ins.0<-mids.1, ins.1<-gbar.0]]
G2 = GATE(nand.eqn)(false,<true,true>) [[out<-r.2, ins.0<-mids.2, ins.1<-mids.3]]
G3 = GATE(c.eqn)(true,<true,true>) [[out<-gbar.0, ins.0<-mids.2, ins.1<-gbar.2]]
G4 = GATE(c.eqn)(true,<true,true>) [[out<-gbar.1, ins.0<-mids.3, ins.1<-gbar.2]]

IMP = ODI({gbar.0,gbar.1}, (ME' |[mids.0,mids.1]| (G0 |[gbar.1,mids.2]| (G1 |[gbar.0,mids.3]|
    (G2 |[mids.2,mids.3]| (G3 |[gbar.2]| G4))))))\{mids.i|i<-\{0..3\}})
    
```

```
channel inside : Events
```

```
Alternate(I,O,P,S,T)
= let WRAPPER(X,Y)
    = ( [] i:inter(I,X) @ i -> inside.i -> WRAPPER(Y,X) )
      [] inside?o:inter(O,X) -> o -> WRAPPER(Y,X)
      [] ( [] i:inter(I,Y) @ i -> BOTTOM )
      [] inside?o:inter(O,Y) -> BOTTOM
    U = union(S,T)
  within DI(inter(I,U),inter(O,U),WRAPPER(S,T))
      [inside.u<->u|u<-U] P
```

```
Alternations(I,O,P,<>) = P
Alternations(I,O,P,<(S,T)>^pairs)
= Alternations(I,O,Alternate(I,O,P,S,T),pairs)
```

HS_TA

```
= Alternations({r.0,r.1,gbar.2},  
               {gbar.0,gbar.1,r.2},TA,  
               <({r.0},{gbar.0}),  
               ({r.1},{gbar.1}),  
               ({r.2},{gbar.2})>)
```

```
assert HS_TA [FD= IMP
```

FDR can be used to verify an asynchronous circuit against a specification in order to

- identify problems with the circuit,
- determine the extent to which it relies on isochronicity, and
- get the specification right.

Functions have been provided that make it simple to

- model the switching behaviour of asynchronous circuits as output-delay-insensitive processes, expressed in CSP_M , and
- separate out concerns when specifying delay-insensitive signalling or handshaking at the interface.