Automatic Verification of Speed-Independent Circuit Designs Using the Circal System

Andrew Bailey

Dept. of Mathematics and Computing Science,
Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven,
The Netherlands.
Tel: +31 40 47 43 18,
Fax: +31 40 43 66 85,
E-mail: amb@win.tue.nl

Abstract. Alain Martin has developed a design method for asynchronous circuits whereby a specification is refined through a series of distinct stages to a gate implementation. The Circal system is able to check equivalence between terms of the Circal process algebra. The system can be used to verify the results of some of Martin's refinements and the gate implementations. It can also be used to analyse the operator network for the necessity of isochronic forks.

1 Introduction

Alain Martin argues for the use of Delay-Insensitive (DI) asynchronous circuits for implementing large scale digital systems [5]. A DI circuit operates correctly regardless of any delay, or variation of delay, in the circuit, whether in a gate or wire, which compared to synchronous design techniques can be seen to be a considerable advantage. Unfortunately, as Martin points out, the class of entirely delay-insensitive circuits is very limited, so he has developed a design method based on Speed-independent techniques. Speed-independent design assumes that delays in gates are arbitrary but that wires have no delay. This assumption is valid until it becomes necessary to fanout a signal to a number of gates. Martin introduces the concept of the isochronic fork to deal with this situation.

Martin's method involves transforming a specification through successive levels of refinement to a circuit implementation. These transformations are supposed to preserve correctness however Smith and Zwarico [10] accuse Martin of being less than rigourous in establishing validity of the transformations and have taken it upon themselves to reconstruct Martin's method. An alternative approach is to map the specification and implementation into a formal system and then establish equivalence between the two as a post design activity. This is the approach taken by Dill [4] using a variant of trace theory.

Dill restricts himself to verifying the final design. Whilst this is very useful, it does mean that the design of the circuit must be completed before verification takes place. Martin's method involves a top-down refinement approach to the design and it would be useful if any errors introduced during refinement were uncovered as early as possible. That is the aim of the verification approach described in this paper.

This paper is structured as follows:

Section 2 describes Circal and the Circal system and introduces the modelling of behaviours.
Section 3 outlines Martin's design method and introduces the L/R process which is the example used throughout.
Section 4 describes how production rules, which are the major intermediate design step, can be modelled in Circal and how sets of production rules can be verified.
Section 5 describes how the operators are modelled and how networks of operators can be verified.
Section 6 describes how the operator networks can be analysed for necessity of isochronic forks.
Section 7 discusses the work presented and areas of further work.

2 Circal and the Circal System

The Circal process algebra was developed by Milne for describing and proving properties of digital systems [8]. It consist of five operators to describe behaviour of a process and three operators to describe the interconnection of a number of processes. The behavioural operators are:

`/\c` – termination or deadlock. This represents a process that can do no further communicating. The `c` indicates that the deadlock is of a particular *sort* (The *sort* of a process is the set of actions to which it can respond).
`(a b)P` – guarding; the process `P` (any term constructed from the behavioural operators) can be executed only after the actions `a` and `b` have occurred simultaneously.
`P + Q` – deterministic choice; this becomes process `P` or `Q` depending on whether the environment in which it operates presents actions that are part of `P` or part of `Q`.
`P & Q` – nondeterministic choice; this is similar to the previous choice operator but its semantic is that of an internal choice i.e., the process itself decides which of `P` and `Q` to offer to the environment.
`P <- (a)Q` – definition; this allows binding of a Circal term to an identifier. Since identifiers can also occur in Circal terms, this enables recursion to be used in describing infinite behaviours.

The structural operators are:

`P*Q` – concurrent composition; this Circal term denotes two process running in parallel, communicating with each other according to the common actions of both.
`P-(a b)` – abstraction; this denotes the process `P` with the actions `a` and `b` removed from its behaviour i.e. the resulting term can no longer synchronise with any external occurrences of these actions.
`P[a/p,b/q]` – relabelling; the process P containing actions `p` and `q` is changed to a process where these actions are renamed as `a` and `b` respectively. This enables the same behaviour to be instantiated for different sets of actions.

A formal semantic is given to each of these operators and a set of algebraic laws are derived. This gives insight into the relationship between the operators e.g. + and & are commutative and associative and distribute through each other, * distributes through & [9]. The most interesting laws enable a structural Circal term to be transformed into a behavioural one. In order to provide an effective, engineer friendly means of using Circal it has been embedded in XTC, a high level, interpreted, general purpose programming language [6]; this language is called XCircal [7].

There are several approaches to modelling when using Circal in hardware design [3]. In this paper *transition based modelling* is used. In this style actions are used to represent changes in value e.g. an action a0 might be used to represented a change for 1 to 0 on a port a which has two possible values. Similarly a1 represents a change for 0 to 1. To demonstrate how logic functions are modelled, and how the features of XTC can be used to ease description, a model for a two input AND function is presented.

```
 1        enum i2_states {S00,S10,S01,S11}
 2
 3        Process AND2(Bool ia,ib,ix, i2_states start_state){
 4                static Bool a,b,x
 5                static Process AND[i2_states]{
 6                  AND[S00] <- (a.1)AND[S10] + (b.1)AND[S01] +
 7                              (a.1 b.1 x.1)AND[S11]
 8                  AND[S10] <- (a.0)AND[S00] + (b.1 x.1)AND[S11] +
 9                              (a.0 b.1)AND[S01]
10                  AND[S01] <- (a.1 x.1)AND[S11] + (b.0)AND[S00] +
11                              (a.1 b.0)AND[S10]
12                  AND[S11] <- (a.0 x.0)AND[S01] + (b.0 x.0)AND[S10] +
13                              (a.0 b.0 x.0)AND[S00]
14                }
15                return (AND[start_state][ia/a,ib/b,ix/x])
16        }
```

Explanation: Lines 3-16 define an XCircal function **AND2** which returns an object of type **Process**. Variables of type **Bool** denote two actions *name*.0 and *name*.1 which are used to denote the changes of value of the physical port *name* in the manner described above. Line 5 declares an array of type **Process** with an index being the enumerated type of line 1. Each element in this type is used to denote the possible combination of values on the input ports. Lines 6-13 are Circal definitions for the behaviour of an AND function. Each definition has all the possible changes of input values given which are dictated by the numerals of the array index. Should the output need to change this occurs simultaneously with the input action(s) that caused it. Thus this is a zero delay model. The four definitions can be regarded as defining a state machine which models the AND behaviour. The initial state is selected in the **return** statement along with the required relabelling as specified in the parameters to the function. e.g. **AND2(p,q,u,S10)** gives an AND behaviour with inputs p and q of starting values 1 and 0 respectively, and output u.

3 An Overview of the Martin Design Method

Martin describes his design method in terms of three levels; source code, object code and VLSI implementations. The source code is based on Hoare's CSP and also uses Martin's concept of the probe [5]. This level of description is subject to manipulation prior to the generation of the object code. These manipulations involve refining the specification into a larger set of concurrent processes that make the object code generation tractable for each process.

As an example Martin's L/R design [5] is used and the schematic is shown in figure 1. With respect to the verification the handshake expansion is used as a specification. The two channels L and R are each implemented by two boolean ports conforming to a four phase handshake protocol. L is *passive* (∘) and R is *active*(•). A passive channel waits for communication to be initiated by the environment in which it operates, whilst with an active channel the circuit is the initiator. It follows that the active channels of a process connect to the passive channels of another. The handshake behaviour for L/R is:

$$*[[li]; ro\uparrow; [ri]; ro\downarrow; [\neg ri]; lo\uparrow; [\neg li]; lo\downarrow]$$

The $*$ means repeat forever the behaviour in parentheses. [*boolexp*] means wait until *boolexp* is true. The semicolon is a sequencing operator, whilst the other expressions are assignments e.g. $li\uparrow$ means li becomes true. It is quite easy to see the difference between the passive and active handshake behaviours of each channel. For L the circuit first waits for the input, li, to become true, whilst the first action on the R channel is to make ro true.

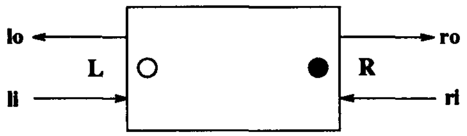


Fig. 1. Schematic for L/R Process

This handshake behaviour maps in to Circal in an obvious way:

```
LR <- li.1 ro.1 ri.1 ro.0 ri.0 lo.1 li.0 lo.0 LR
```

Martin's object code consists of a set of production rules. These have the following form:

$$G \mapsto S$$

where G is a boolean expression, called the guard, and S an unordered list of simple assignments. If G is true the rule 'fires' and the assignments will take place (However, no timing is given, all that can be said is that the assignments will eventually take place).

Example: $a \wedge b \mapsto c \uparrow$

Explanation: if a and b are true then c will become true (after some arbitrary amount of time).

Several properties of these production rules are required in order that a circuit may eventually derived from them.

- Stability - G is either false or remains true until S has completed.
- Noninterference - Complimentary production rules (i.e. $G1 \mapsto x \uparrow$ and $G2 \mapsto x \downarrow$) must not execute at the same time. This is assured by requiring that $\neg G1 \vee \neg G2$ holds invariantly.

Once a suitable set of production rules has been devised they can be manipulated, using global invariants for example, to yield an operator implementation. Operators are the basic building blocks having the behaviour of combinational functions or simple sequential circuits such as registers or Muller-C elements. The final stage is to analyse the production rules to determine where it is necessary to insert isochronic forks.

4 Verifying Production Rules

In order to validate the production rules they must be first modelled in Circal. In Martin's method each production rule operates in parallel so the natural method of modelling would be to have one or more processes to model each rule. It has already been shown how to model logic functions which can be used in the construction of the G term, so a model for the $\mapsto$ must be constructed. The key requirement for proper behaviour is that of stability of the guard. Since the model that is to be constructed is a computation it must detect and signal if the stability requirement is violated.

Circal model for $\mapsto$

```
Process TRANS(Bool ia, Event ix){
        static Bool a
        static Event x
        static Process TRAN,/\t(a.0 a.1 x){
          TRAN <- (a.1)((x)(a.0)TRAN + (a.0)/\t + (x a.0)/\t)
          }
        return (TRAN[ia/a,ix/x])
}
```

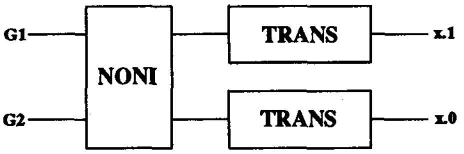
Explanation: The `Bool` identifier `a` is the guard and the `Event x` is the assignment. If the guard becomes true (`a.1`) then `TRAN` evolves into the inner parenthesised term. The behaviour has then three possibilities;(i) the assignment takes place (`x`) and then the guard becomes false (`a.0`); (ii) the guard becomes false (`a.0`); (iii) the guard becoming false and the assignment occur at the same time. In the last two cases the behaviour terminates after the actions. Termination is used to flag violation of the stability criteria.

The other major criteria is that of non-interference between complimentary pairs of production rules. Another process, `NONI` is defined which connects to pairs of `TRANS` processes as shown in figure 2. It monitors pairs of guards and filters out vacuous firings. A vacuous firing occurs when a guard is enabled to cause a variable to be assigned a value that it already has. Should such firings be passed to the **TRANS** process it will try to produce the appropriate output. This means that two **x.0**s might occur without an intervening **x.1**. This violates the modelling method.

```
Process NONI(Bool ia,ib,ix,iy, i3_states start_state){
     static Bool a,b,x,y
     static Process NI[i3_states],/\n(a b x y){

     NI[S000] <- (a.1 x.1)NI[S100] + (b.1)NI[S010] + (a.1 b.1)/\n
     NI[S100] <- (a.0 x.0)NI[S001] + (b.1)/\n + (a.0 b.1 x.0 y.1)NI[S011]
     NI[S010] <- (a.1)/\n + (b.0)NI[S000] + (a.1 b.0 x.1)NI[S100]
     NI[S001] <- (a.1)NI[S101] + (b.1 y.1)NI[S011] + (a.1 b.1)/\n
     NI[S011] <- (a.1)/\n + (b.0 y.0)NI[S000] + (a.1 b.0 x.1 y.0)NI[S100]
     NI[S101] <- (a.0)NI[S001] + (b.1)/\n + (a.0 b.1 y.1)NI[S011]
     }
     return NI[start_state][ia/a,ib/b,ix/x,iy/y]
}
```

Explanation: The **Bool** variables **a** and **b** denote the changes of values of the guards and **x** and **y** are used to trigger the **TRANS** processes as appropriate. If the non-interference criteria is violated the process terminates. The last digit denotes the value of the variable to which the production rules are assigning values. Vacuous firings are detected and absorbed by detecting whether the output is already at the value required by the guard becoming true.



```
Process PR00(Bool g1,g2,x){
    Bool xx,yy
    return (NONI(g1,g2,xx,yy,S000)*
            TRANS(xx,x.1)*
            TRANS(yy,x.0) -(xx yy))
}
```

Fig. 2. Processes for Modelling Production Rules

For the L/R element the following production rules are generated by Martin's design method. Note that these are in what Martin calls program order, and that a state variable, x, has been introduced to get the required functionality. Bringing together the complementary pairs an XTC script can be constructed to model these in Circal.

$$\begin{aligned}
\neg x \wedge li &\mapsto ro\uparrow \\
ri &\mapsto x\uparrow \\
x &\mapsto ro\downarrow \\
x \wedge \neg ri &\mapsto lo\uparrow \\
\neg ri &\mapsto x\downarrow \\
\neg x &\mapsto lo\downarrow
\end{aligned}$$

```
Process PR[4]
Bool li,lo,ri,ro,x
Bool a,b,c,d,e,f
Bool g1,g2,g3,g4,g5,g6

PR[1] <- NOT(x,a,S0)*AND2(a,li,g1,S10)*
         PR00(g1,x,ro) - a

PR[2] <- NOT(ri,a,S0)*AND2(x,a,g3,S01)*
         NOT(x,g4,S0)*
         PR01(g3,g4,lo) - a

PR[3] <- NOT(li,g6,S0)*PR01(ri,g6,x)
```

Unfortunately, as they stand, the behaviour of the production rules above can not be shown equivalent to the specification. The reason may be deduced from examining the differences in style between the specification and the AND model. In each Circal definition for the AND model all possible input changes are allowed; it has the property that Dill calls *receptiveness* [4]. In contrast the inputs of the specification only change in accordance with the four phase handshake protocol. In order to carry out the verification this difference must be resolved. This is accomplished by providing a process or processes that model an environment for the production rules.

Two constraints are used to construct environments for this type of production rule; one for each type of channel.

Constraint for passive channel

```
Process PASSIVE_CON(Bool req, ack){
        static Bool a
        static Bool b
        static Process CONS1,CONS2,CONS3,CONS4,/\c(a b){
           CONS1 <- (a.1)CONS2 + (b.1)/\c + (a.1 b.1)/\c
           CONS2 <- (b.1)CONS3
           CONS3 <- (a.0)CONS4 + (b.0)/\c + (a.0 b.0)/\c
           CONS4 <- (b.0)CONS1
        }
        return (CONS1[req/a,ack/b])
}
```

Explanation: From the relabelling in the return statement it can be seen that a connects to the request port and b to the acknowledge. The required handshake behaviour is seen by examining the first guard in each of the definitions. Where a choice exists this is to fulfill the receptiveness criteria, however should it occur then this constitutes an error, hence /\c.

Constraint for active channel

```
Process ACTIVE_CON(Bool req, ack){
        static Bool a
        static Bool b
        static Process CONS1,CONS2,CONS3,CONS4,/\c(a b){
           CONS1 <- (a.1)CONS2
           CONS2 <- (b.1)CONS3 + (a.0)/\c + (a.0 b.1)/\c
           CONS3 <- (a.0)CONS4
           CONS4 <- (b.0)CONS1 + (a.1)/\c + (a.1 b.0)/\c
        }
        return (CONS1[req/a,ack/b])
}
```

Explanation: Similar to above but the request behaviour should be receptive.

Now it is possible to formulate a model which should be equivalent to the Circal specification for L/R.

```
PR_BEH <- ~(PASSIVE_CON(li,lo)*
            ACTIVE_CON(ro,ri)*
            PR[1]*PR[2]*PR[3] - (x g1 g3 g4 g6))
```

Explanation: The composition represents the production rules and environment executing in parallel, whilst the abstraction hides the state variable and the guard values. The ~ applies the laws of Circal to generate a behaviour from a structure which can then be checked for equivalence with the specification.

5 Verification of Operator Implementation

Once a satisfactory set of production rules have been arrived at, a network of operators can be derived. The behaviour of an operator is also specified by production rules, for example the AND operator behaviour is:

$$a \wedge b \mapsto x \uparrow$$
$$\neg a \vee \neg b \mapsto x \downarrow$$

Thus operators can be modelled in Circal using the models already described, however if an operator is supposed to have the behaviour of a common logic function then it might be better to use the direct model of the behaviour, such as presented in section 2, rather than the production rules. This can be achieved using the following process that can be composed with the logic function models.

```
Process CDELAY(Bool ia,ix, i2_states start_state){
        static Bool a,x
        static Process CD[i2_states],/\t(a x){
                CD[S00] <- (a.1)CD[S10]
                CD[S10] <- (x.1)CD[S11] + (a.0)/\t  + (a.0 x.1)/\t
                CD[S01] <- (x.0)CD[S00] + (a.1)/\t  + (a.1 x.0)/\t
                CD[S11] <- (a.0)CD[S01]
                }
        return(CD[start_state][ia/a,ix/x])
}
```

Explanation: This behaviour is that of a causal delay, and is similar to that of **TRANS**, but **x** is now a **Bool**. As with **TRANS**, if the input changes before the previous value has propagated to the output then this constitutes an error in the design hence the process terminates.

There are two state holding operators commonly used, the Muller-C and the flip-flop. In the implementation of the L/R element both can be used, here the flip-flop implementation is described. For the flip-flop the production rule specification is:

$$\begin{aligned} x &\mapsto z\uparrow \\ \neg y &\mapsto z\downarrow \end{aligned}$$

In fact the behaviour is that of an RS flip-flop but with the reset being active low. Using this observation it is easiest to create a model for an RS flip-flop and use the **NOT** process to invert the reset input.

```
Process F2(Bool is,ir,ib, i3_states start_state){
        static Bool s,r,b
        static Process F[i3_states],/\f(s r b){
         F[S000] <- (s.1 b.1)F[S101] + (r.1)F[S010] + (s.1 r.1)/\f
         F[S010] <- (r.0)F[S000] + (s.1)/\f + (r.0 s.1 b.1)F[S101]
         F[S101] <- (s.0)F[S001] + (r.1)/\f + (s.0 r.1 b.0)F[S010]
         F[S001] <- (s.1)F[S101] + (r.1 b.0)F[S010] + (s.1 r.1)/\f
        }
        return F[start_state][is/s,ir/r,ib/b]
}
```

The model for the flip-flop is generated by composing this model with the **CDELAY** and **NOT** models.

An operator network for the L/R process is show in figure 3. An alternative network is formed by replacing the flip-flop with a Muller-C. Capturing this network in Circal is simply a matter or instantiating the requisite operator models with appropriate actions. As with the production rules it is necessary to include the constraints in order to show testing equivalence between the operator network and the specification.

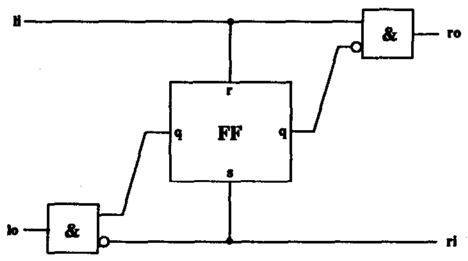


Fig. 3. Flip-Flop Implementation of L/R

6 Isochronic Forks

The operator implementation assumes that all delay is associated with the operators and none with the wires. However if an output is fanned out to several operators then problems might possibly arise. Consider figure 4(a); this represents a circuit where as soon as the output of the CDELAY changes the value is propagated to each of the F processes. This is what Martin calls an isochronic fork which means that the signal should arrive at all it's destinations simultaneously. All forks in the operator implementations are isochronic which may not be necessary. Martin analyses the production rules to check where isochronic forks are necessary. However it is possible to use Circal and the Circal system diagnostics to perform this analysis on the operator implementation.

The method is as follows: each CDEALY that is connected to more than one F type process is replaced by a WIRE_FORK as shown in figure 4(b). The behaviour of the WIRE_FORK is as follows:

```
Process WIRE_FORK(Bool ia,ix,iy, il_states start_state){
        static Bool a,x,y
        static Process WF[il_states],/\w(a x y){
        WF[S0] <- (a.1)((x.1)((y.1)WF[S1] + (a.0)/\w + (y.1 a.0)/\w) +
                       (y.1)((x.1)WF[S1] + (a.0)/\w + (x.1 a.0)/\w) +
                       (x.1 y.1)WF[S1] +
                       (a.0)/\w + (x.1 y.1 a.0)/\w)

        WF[S1] <- (a.0)((x.0)((y.0)WF[S1] + (a.1)/\w + (y.0 a.1)/\w) +
                       (y.0)((x.0)WF[S1] + (a.1)/\w + (x.0 a.1)/\w) +
                       (x.0 y.0)WF[S1] +
                       (a.1)/\w + (x.0 y.0 a.1)/\w))
        }
        return WF[start_state][ia/a,ix/x,iy/y]
}
```

Explanation: Each definition represents the current input value. Only one action can occur but this guards a choice of actions. The first three represent the possible sequences on the two outputs. The first two represent one branch being faster than the other so the process must then generate the appropriate action for the slower branch. In each choice the model is receptive to a change on the input but if this occurs then, as before, this is considered an error so the process terminates.

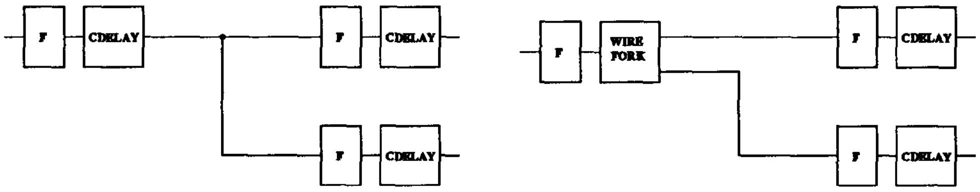


Fig. 4. Implementation with (a) Isochronic fork (b) Wire fork

Having introduced WIRE_FORKS the operator implementation is checked for equivalence against the specification. If they are equivalent then all forks in the circuit maybe implemented with wire forks. If the check fails then one or more of the forks need to be implemented either as isochronic or asymmetric isochronic forks. The asymmetric isochronic fork guarantees that one branch of the fork is always faster than the other. By examining the diagnostics from the system it may be possible to determine which forks this applies to. It is also possible to systematically substitute the isochronic fork models for wire fork models.

For the L/R circuit there are two forks, both on inputs. Introducing the wire forks into the circuit the equivalence check fails. The relevant part of the diagnostics are produced here.

```
Failed after following trace: li.1 ro.1 ri.1
SPEC = ro.0
LR_IMP_FORKS = /\ & ((ro.0 lo.1) + ro.0 + lo.1) & ro.0

Failed after following trace: li.1 ro.1 ri.1 ro.0 ri.0 lo.1 li.0 lo.0
SPEC can do li.1
LR_IMP_FORKS can do no actions
```

Explanation: The check fails because the two processes behave differently after the indicated trace. LR_IMP_FORKS can clearly do more things than the specification as shown by the regenerated behaviour. First there is a non-deterministic choice between three deterministic behaviours. The /\ indicates that the constraint on the flip-flop has been violated. Recall that this occurs if s is 1 and r is 0; this condition arises because the wire fork allows the signal to reach the AND before it reaches the flip-flop. By the handshake protocol this causes the input on ri which can then propagate to s. All of this can occur before the li signal reaches r (due to the unbounded delay in the fork model). Clearly that fork cannot be a wire fork In fact in turns out that it must be asymmetric with the faster branch being connected to the flip-flop. The second trace indicates violation of the flip-flop constraint by the symmetrical case to the one discussed above.

7 Discussion

The work described here supersedes the previous work using Circal in asynchronous circuit design [2] [1]. In the first reported work it was found that the approach to verification was inadequate. This lead to the formulation of verification using constraints as described here and first reported in the second work. Upon critically examining that work I decided that the operator models were in adequate and so set about reformulating them. Subsequently I discovered that this reformulation coincided with Dill's concept of receptiveness, thus lending support to my new operator models.

There are two major differences between the approach described here and that of Dill with respect to models and use. Here operators models are constructed from two processes; the CDELAY and a process that models the operator function. This hopefully leads to less errors in the models and increasing confidence that when in-equivalence arises it is due to a design error. Secondly, rather than applying the verification to the final design, I have shown how Circal can be applied to the verification of the production rules. Again a constructive approach to the models was taken, separating out the active assignment using **TRANS** from the detection of computation interference using **NONI**.

As well as the example here these models and the method have been applied to Martin's distributed mutual exclusion design and the Handshake circuits from the previous work have been verified using the new operator models.

Acknowledgements

Thanks go to Ad Peeters for his comments on the drafts of this paper and the Eindhoven VLSI Club for listening and commenting on a presentation of this material. I also acknowledge HardLab, Dept. of Computer Science, University of Strathclyde for allowing me to use the Circal system.

References

1. A. Bailey and George Milne. Verifying the correctness of asynchronous design modules. Technical Report HDV-23-92, University of Strathclyde, Department of Computer Science, Glasgow, February 1992.
2. A. Bailey and G.J. Milne. Verification of Philips' Operators for VLSI Programming. Technical Report HDV-17-91, University of Strathclyde, Glasgow, August 1991.
3. Andrew Bailey. *Modelling, design and analysis of digital circuits using Circal.* PhD thesis, University of Strathclyde, *forthcoming in* 1993.
4. D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT Press, 1989.
5. Alain J. Martin. Programming in VLSI. In C.A.R. Hoare, editor, *Developments in Communication and Concurrency*, pages 1–64. Addison-Wesley, 1989.
6. G.A. McCaskill. The XTC language reference manual. Technical Report HDV-14-91, Univeristy of Strathclyde, Department of Computer Science, 1991.
7. G.A. McCaskill. XCircal: Users' guide and reference manual. Technical Report HDV-18-91, University of Strathclyde, Department of Computer Science, Glasgow, October 1991.
8. G.J. Milne. Circal and the representation of communication, concurrency and time. *ACM Trans. on Programming Languages and Systems*, 7(2), 1985.
9. F. Moller. The semantics of Circal. Technical Report HDV-3-89, University of Strathclyde, Department of Computer Science, Glasgow, Scotland, April 1989.
10. Scott F. Smith and Amy E. Zwarico. Provably correct synthesis of asynchronous circuits. In J. Staunstrup and R. Sharp, editors, *Designing Correct Circuits*, pages 237–260. North-Holland, 1992.