Applying the Concurrency Workbench to the Verification of DI Circuits

Hemangee K. Kapoor and Mark B. Josephs
SCISM, South Bank University, London
{kapoorhk, josephmb}@sbu.ac.uk

1 Introduction

On-chip modules that interact through "delay-insensitive" (DI) interfaces (such as handshaking ports) are becoming attractive to system designers. There are several reasons for this:

1. *Necessity*. With multiple clock domains or clockless logic, there need not be a common clock on which modules can synchronise.

2. *Convenience*. Re-use of modules in different designs and in different implementation technologies is facilitated by the removal of timing constraints.

3. *Robustness*. In deep sub-micron CMOS technology wire delays dominate over gate delays.

In this extended abstract, we shall consider the modelling of such delay-insensitive modules and the verification of circuits that are composed out of them. More specifically, we consider some small asynchronous controllers and verify their implementation using a popular verification tool, the Concurrency Workbench (CWB) [1, 12].

The CWB is an automated tool that helps in the manipulation and analysis of concurrent system specifications [1]. A variety of equivalence relationships are supported including testing equivalence [4]. Model checking can also be performed. The CWB has been applied in modelling and verification of asynchronous circuits and microprocessors [9, 15] before, but the property of delay-insensitivity has not been considered. The main modelling language used by the CWB is the process calculus CCS [11].

The remaining sections are organised as follows. In section 2 we explain how to model DI modules in CCS. Section 3 describes the modelling language DI-Algebra. The translation procedure from DI-Algebra to CCS supported by the tool di2ccs is described in section 4. Section 5 presents an example of verification, before we conclude. Appendix A describes the MUST-testing preorder.

2 Defining DI Processes in CCS

A DI module communicates with its environment through wires of unbounded delay. It is considered erroneous for two transitions to be propagating along a wire at the same time; this is known as transmission interference [16, 17]. Such a wire can be modelled in CCS as follows:

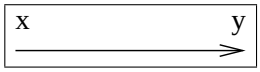

Figure 1 Wire with input terminal x and output terminal y.

$\mathtt{W} = \mathtt{x}\,.\,(\mathtt{'y}\,.\,\mathtt{W} + \mathtt{x}\,.\,@)$,

where actions x and 'y model the input and output of transitions at the corresponding terminals, and divergent process @ models interference, Figure 1. Thus, safe usage of a wire requires that input and output alternate.

The connection of two wires in series should behave like a single wire. Unfortunately, this is not the case under the standard equivalence (bisimulation) of CCS. We shall therefore adopt a semantic model in which this equivalence does hold, namely, the failures/divergences model of CSP [5] or, equivalently, the MUST-testing preorder [4] (see Appendix), where the behaviour of a divergent process is considered to be undefined. (Note that the divergent process @ of CCS is called CHAOS in CSP.)

By attaching a wire to each terminal of a process P, we can construct a DI version Di_P of P, Figure 2.

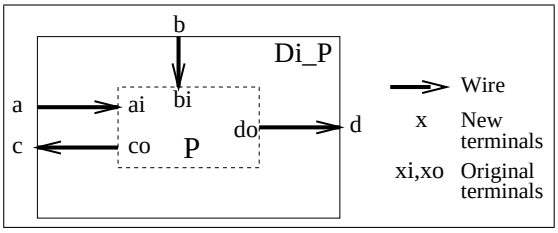

Figure 2 Delay-Insensitive model (Di_P) for P, with original terminals renamed and hidden

Example

Consider the following two processes in CCS:

$\mathtt{P} = \mathtt{a}\,.\,\mathtt{b}\,.\,\mathtt{'c}\,.\,\mathtt{P}$

$\mathtt{Q} = \mathtt{b}\,.\,\mathtt{a}\,.\,\mathtt{'c}\,.\,\mathtt{Q}$

The two processes P and Q are not equivalent, but their delay-insensitive versions Di_P and Di_Q are equivalent under MUST-testing, as can be seen using the CWB:

$\mathtt{musteq}(\mathtt{P}, \mathtt{Q}) = \mathtt{false}$

$\mathtt{musteq}(\mathtt{Di\_P}, \mathtt{Di\_Q}) = \mathtt{true}$

In general, a process P is defined to be delay-insensitive if Di_P is equivalent to P. This is known as the Foam Rubber Wrapper postulate [13, 16]. In particular, Di_P is delay-insensitive since Di_Di_P is equivalent to Di_P.

3 The Modelling Language DI-Algebra

One approach to the modelling of a DI module is to describe it as a process P in CCS and then verify that P is delay-insensitive, as defined above. An alternative is to use a language in which only DI processes can be described; DI-Algebra [8], a variant of CSP, is such a language. Moreover, the denotational semantics [6, 10] of DI-Algebra is compatible with the failures/divergences model of CSP, so processes can still be characterised by MUST-testing. The algebra also has a complete set of algebraic laws [3].

We have adopted the following concrete syntax for DI-Algebra:

$$
\begin{array}{lll}
declaration & ::= & id = proc \\
proc & ::= & highproc \mid lowproc \\
highproc & ::= & id\ [\ ||\ id]* \\
lowproc & ::= & inputs\ ,\ outputs\ stmt \\
guard & ::= & sig? \mid sig! \mid \texttt{skip} \\
stmt & ::= & \\
 & & \texttt{CHAOS} \mid \texttt{STOP} \mid id \\
 & & \mid stmt\ /\ sig? \mid guard\ ;\ stmt \\
 & & \mid [\ guard \rightarrow stmt\ [\#\ guard \rightarrow stmt\ ]*\ ] \\
 & & \mid stmt\ \texttt{ND}\ stmt \mid (stmt)
\end{array}
$$

Here $inputs$ and $outputs$ are (disjoint) input and output alphabets, respectively. In the parallel composition ($P\ ||\ Q$) of two processes P and Q, the input alphabet of P should be disjoint from that of Q; likewise the output alphabet of P should be disjoint from that of Q. Note that P ND Q denotes a non-deterministic choice between P and Q, whereas $[\ g_1 \rightarrow P_1\ \#\ g_2 \rightarrow P_2\ ]$ denotes a guarded choice.

The advantages of using DI-Algebra rather than CCS for modelling DI modules are as follows:

1. There is no need to verify that a process is DI.
2. Point to point connection is directly modelled by the parallel composition operator "||" of DI-Algebra, rather than by the combination of "|" and "\" required by CCS.
3. The after-input operator of DI-Algebra is convenient for defining the behaviour of modules, especially their initial state.
4. There is a simple translation from processes in DI-Algebra into Petri nets [7] from which asynchronous logic can be synthesised using the tool Petrify [2].

Of course, in order to verify designs modelled in DI-Algebra using the Concurrency Workbench, we first need to translate them into CCS. This translation procedure is automated by our tool *di2ccs*, which is discussed next.

4 The translation tool di2ccs

The translation of processes from DI-Algebra into CCS has been automated in a tool *di2ccs* (implemented in Java).

4.1 Translation of "low" processes

The major tasks done by the tool are:

- Parsing of process declarations in DI-Algebra.
- Application of transformation rules given below to generate corresponding declarations in CCS.
- Declaration of DI versions of the above processes.

For example, the process P declared in DI-Algebra by
$P = \{a, b\}, \{c\}\ a?\ ;\ b?\ ;\ c!\ ;\ P$
is syntactically transformed to the CCS process
$P = a\ .\ b\ .\ 'c\ .\ P$
To preserve its semantics, wires are then attached:
$Di\_P =$
$(\ PBW[a/x, pa/px, ai/y]\ |$
$\ PBW[b/x, pb/px, bi/y]\ |$
$\ W[co/x, c/y]\ |$
$\ P[ai/a, bi/b, co/c]\ )\backslash\{ai, pa, bi, pb, co\}$

4.1.1 Syntactic transformation rules on guards and statements

- $x? \Rightarrow x$
- $x! \Rightarrow$ *'x*
- skip $\Rightarrow$ tau
- CHAOS $\Rightarrow$ @
- STOP $\Rightarrow 0$
- $P/x? \Rightarrow$ *'px* . P
- $g\ ;\ P \Rightarrow g\ .\ P$
- $[\ g_1 \rightarrow P_1\ \#\ \ldots\ \#\ g_n \rightarrow P_n\ ]$
  $\Rightarrow g_1\ .\ P_1\ +\ \ldots\ +\ g_n\ .\ P_n$
- P ND $Q \Rightarrow$ tau . P + tau . Q

This just leaves us to consider the connection of wires to terminals.

4.1.2 Pushback wires

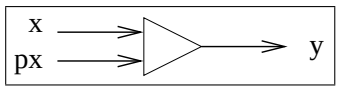

Figure 3 Pushback wire with input terminals x and px and output terminal y

Since the after-input operator $P/x?$ is translated into *'px . P*, instead of attaching a wire to x, we use a "pushback" wire, Figure 3, defined by

$\texttt{PBW} = \texttt{x} \,.\, \texttt{PBW}' + \texttt{px} \,.\, \texttt{PBW}'$

$\texttt{PBW}' = \,'\texttt{y} \,.\, \texttt{PBW} + \texttt{x} \,.\, @ + \texttt{px} \,.\, @$

That is, actual input x and pushed back input px are merged.

4.2 Translation of "high" processes

In the case of parallel composition, the alphabets of the processes being composed have to be analysed. Let P and Q be two processes composed in parallel with input (output) alphabets $A1$ ($B1$) and $A2$ ($B2$) respectively. Let *internals* be defined as the set of shared signals between P and Q, as follows: $internals = (A1 \cap B2) \cup (A2 \cap B1)$. This gives us the translation

$P \,||\, Q \Rightarrow (Di\_P \mid Di\_Q) \backslash internals$

where Di_P and Di_Q are delay-insensitive CCS translations of process P and Q respectively.

4.2.1 Optimisation

The CWB builds and analyses a transition system representation of a process. We can observe from the above translations that, if the size (number of states) of a process P is S_P, then the size of Di_P has an upper bound of $S_P \times 3^n$, where n is the total number of signals in the alphabets of process P. Note that the size of both W and PBW is 3, so far as the CWB is concerned.

If we want to verify a possible implementation involving several components composed in parallel, the size of the implementation increases multiplicatively with the number of components. To reduce this increase in the size (and make large circuits verifiable), we have optimised our tool to generate fewer wires. In the case of parallel composition of two processes P and Q, instead of composing Di_P with Di_Q which would have a W and a PBW per internal signal, we generate a single PBW. Thus there is a reduction of size by a factor of 3^m, where m is the number of internal signals.

4.2.2 Example

Consider the following declarations in DI-Algebra:

$P = \{a, b\}, \{c\}\ a?\ ;\ b?\ ;\ c!\ ;\ P$

$Q = \{c\}, \{d, e\}\ c?\ ;\ d!\ ;\ e!\ ;\ Q$

$M = P \,||\, Q$

Syntactic transformation of "low" processes P and Q into CCS gives:

$P = a \,.\, b \,.\, 'c \,.\, P$

$Q = c \,.\, 'd \,.\, 'e \,.\, Q$

The "high" process M is then transformed using the parallel combination of P and Q along with the attached wires. Note that only one pushback wire is generated for the internal signal c.

$Di\_M =$

$(PBW[a/x, pa/px, ai/y] \mid$
$\ PBW[b/x, pb/px, bi/y] \mid$
$\ PBW[co/x, pc/px, ci/y] \mid$
$\ W[do/x, d/y] \mid W[eo/x, e/y] \mid$
$\ P[ai/a, bi/b, co/c] \mid Q[ci/c, do/d, eo/e]$
$) \backslash \{ai, pa, bi, pb, ci, pc, co, do, eo\}$

5 Examples of Verification

In verifying an implementation I against a specification S, we need only check that I refines S, i.e. $\texttt{mustpre}(\texttt{S}, \texttt{I})$.

5.1 Verification of Call element

A call element can be declared in DI-Algebra as follows:

$Call = \ [\ a0? \rightarrow c!; C0 \ \#\ a1? \rightarrow c!; C1\ ]$

$C0 = \ [\ b? \rightarrow d0!; Call$
$\qquad \#\ a0? \rightarrow \textsf{CHAOS} \ \#\ a1? \rightarrow \textsf{CHAOS}\ ]$

$C1 = \ [\ b? \rightarrow d1!; Call$
$\qquad \#\ a0? \rightarrow \textsf{CHAOS} \ \#\ a1? \rightarrow \textsf{CHAOS}\ ]$

An implementation [18] of a Call element is shown in Figure 4. Even with the optimisation of section 4.2.1, the CCS model generated by di2ccs has almost 10^7 states. As it was not possible to verify this directly on our machine (a 1.4 GHz Pentium 4 with 256 MB RAM), we adopted a hierarchical approach. We divided the circuit into two components, an abstract description P (of the component shown dotted in the figure) and the latch element.

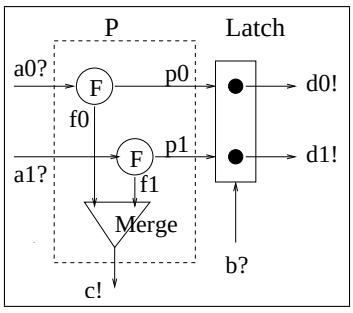

Figure 4 Call Element

$P = \ [\ a0? \rightarrow c!; p0!; P \ \#\ a1? \rightarrow c!; p1!; P\ ]$

$Latch = \ [\ \ p0? \rightarrow [\ p1? \rightarrow \textsf{CHAOS}$
$\qquad \#\ b? \rightarrow d0!; Latch\ ]$
$\qquad \#\ p1? \rightarrow [\ p0? \rightarrow \textsf{CHAOS}$
$\qquad \#\ b? \rightarrow d1!; Latch\ ]\ ]$

$M = \ P \,||\, Latch$

We first verified P in parallel with the latch against the specification of the Call element, as follows:
Size of Di_CALL = 5833, Size of Di_M = 196857
$\texttt{mustpre}(\texttt{Di\_CALL}, \texttt{Di\_M}) = \texttt{true}$
Time taken = 30 min, 36 sec

P is implemented as two forks and a merge as shown below:

$Fork0 = \quad a0?; p0!; f0!; Fork0$
$Fork1 = \quad a1?; p1!; f1!; Fork1$
$Merge = \quad [\, f1? \rightarrow c!; Merge$
$\quad\quad\quad \#\ f0? \rightarrow c!; Merge\,]$
$N = \quad\quad Fork0 \,||\, Fork1 \,||\, Merge$

We then verified the implementation of P.
Size of Di_P = 1216, Size of Di_N = 39375
$\texttt{mustpre}(\texttt{Di\_P}, \texttt{Di\_N}) = \texttt{true}$
Time taken = 19.596 sec

Table 1 shows verification results for some other circuits given in [18].

6 Conclusion

We have defined a method to verify delay-insensitive processes using the CWB. As CCS is its main modelling language, we have added a front-end *di2ccs* that translates from DI-Algebra into CCS. This involves attaching a pushback wire to each input terminal and a wire to each output terminal of a process. Delay-insensitive circuits can thus be automatically verified using the tools di2ccs and CWB.

As observed from experiments, using this method the CWB suffers from the state explosion problem even for small circuits. Hierarchical verification can help here, but alternative tools are also worth investigating.

Specification (S)	Size of S	Implementation (I)	Size of I	Time (sec)
OR	190	XOR	82	0.084
Connector	325	OR || Mixer	1143	0.445
Connector	325	PAR || Join	1143	0.452
Mod-3 Counter	163	Mod-1 Counter || Join || Forks || Merge || Toggle	1596	0.775
Duplicator	487	PAR || Mixer	2358	0.827
2-to-4 Converter	568	Merge || Toggle	1953	0.851
Sequencer	973	Mixer || Join	6588	2.828
Latch	1459	Non-Receptive Mixer || Join	6102	2.844

Table 1: Performance of the CWB on various DI circuits

References

[1] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics-based Verification Tool for Finite-state Systems. *Proceedings of the Workshop on Automated Verification Methods for Finite-state Systems, Lecture Notes in Computer Science*, 407, 1989.

[2] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. *IEICE Transactions on Information and Systems*, 3(E80-D):315–325, 1997.

[3] Rix Groenboom, Mark B. Josephs, Paul G. Lucassen, and Jan Tijmen Udding. Normal Form in a Delay-Insensitive Algebra. *Proceedings of the IFIP Transactions on Asynchronous Design Methodologies*, pages 57–70, April 1993.

[4] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[5] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science, 1985.

[6] Mark B. Josephs. Receptive Process Theory. *Acta Informatica*, 29(1):17–31, 1992.

[7] M.B. Josephs and D.P. Furey. Delay-insensitive interface specification and synthesis. *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 169–173, March 2000.

[8] M.B. Josephs and J.T. Udding. An overview of D-I algebra. *System Sciences, 1993, IEEE Proceeding of the Twenty-Sixth Hawaii International Conference*, 1:329 – 338, January 1993.

[9] Ying Liu. *AMULET1 Specification and Verification in CCS*. PhD thesis, Department

of Computer Science, Univeristy of Calgary, September 1995.

[10] Paul G. Lucassen. *A Denotational Model and Composition Theorems for a Calculus of Delay-Insensitive Specifications*. PhD thesis, Dept. of C.S., Univ. of Groningen, The Netherlands, May 1994.

[11] R. Milner. *Communication and Concurrency*. Prentice-Hall International Series in Computer Science, 1989.

[12] Faron Moller and Perdita Stevens. Edinburgh Concurrency Workbench user manual (version 7.1). Available from http://www.dcs.ed.ac.uk/home/cwb/.

[13] Charles E. Molnar, Ting-Pien Fang, and Frederick U. Rosenberger. Synthesis of delay-insensitive modules. *In Henry Fuchs, editor, 1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86, 1985.

[14] Roberto Segala. Quiescence, fairness, testing and the notion of implementation. *Information and Computation*, 138:194–210, 1997.

[15] Kenneth S. Stevens. *Practical Verification and Synthesis of Low Latency Asynchronous Systems*. PhD thesis, Department of Computer Science, University of Calgary, September 1994.

[16] Jan Tijmen Udding. A Formal Model for Defining and Classifying Delay-Insensitive Circuits . *Distributed Computing*, 1(4):197–204, 1986.

[17] Jan L. A. van de Snepscheut. Trace Theory and VLSI Design . *LNCS*, 200, 1985.

[18] Tom Verhoeff. Encyclopedia of Delay-Insensitive Systems (EDIS). http://edis.win.tue.nl/edis.html.

A MUST-testing

After a process p has engaged in a trace s, it may be in one of several possible states, in each of which a set of actions is enabled. $\mathcal{A}(p,s)$ denotes the set of such so-called Acceptance sets. We write $\mathcal{A} \subset\subset \mathcal{B}$ if for every Acceptance set $X \in \mathcal{A}$ there exists some Acceptance set $Y \in \mathcal{B}$ such that $Y \subseteq X$. After s, if process p has infinite internal computation, it is said to be *divergent*; otherwise it is said to be *convergent*, denoted by $p \downarrow s$. The MUST testing preorder is then defined [4] as follows:
$\texttt{mustpre}(p,q)$ if, for every sequence s of actions $p \downarrow s$ implies

i) $q \downarrow s$, and

ii) $\mathcal{A}(q,s) \subset\subset \mathcal{A}(p,s)$

Thus, $\texttt{musteq}(p,q)$ iff $\texttt{mustpre}(p,q)$ and $\texttt{mustpre}(q,p)$.

Note that the idea of using MUST-testing in the context of I/O automata was considered in [14], but divergence was not considered.

A.1 Two wires in series is equivalent to a single wire

Figure 5 shows that two wires in series is equivalent to a single wire. Figure 6 shows that a single wire in series with a pushback wire is equivalent to a pushback wire.

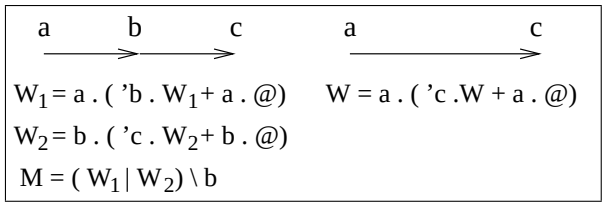

Figure 5 $\texttt{musteq}(\texttt{M},\texttt{W}) = \texttt{true}$.

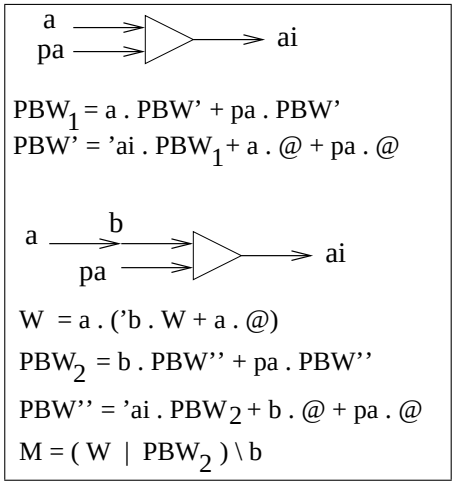

Figure 6 $\texttt{musteq}(\texttt{M},\texttt{PBW}_1) = \texttt{true}$.