Report Number: WUCS-89-54

1990-03-23

An Algebra for Delay-Insensitive Circuits

Authors: Mark B. Josephs and Jan Tijmen Udding Washington University in St. Louis

A novel process algebra is presented; algebraic expressions specify delay-insensitive circuits in terms of voltage-level transitions on wires. The approach appears to have several advantages over traditional state-graph and production-rule based methods. The wealth of algebraic laws makes it possible to specify circuits concisely and facilitates the verification of designs. Individual components can be composed into circuits in which signals along internal wires are hidden from the environment.

AN ALGEBRA FOR DELAY-INSENSITIVE CIRCUITS

Mark B. Josephs
Jan Tijmen Udding

WUCS-89-54

March 1990

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

An Algebra for Delay-Insensitive Circuits

Mark B. Josephs
Programming Research Group
Oxford University Computing Laboratory
11 Keble Road, Oxford OX1 3QD, U.K.
Phone: +44-865-272574
E-mail: mark%prg.oxford.ac.uk@nss.cs.ucl.ac.uk

Jan Tijmen Udding
Department of Computer Science
Washington University
Campus Box 1045, St. Louis, MO 63130, U.S.A.
Phone: 314-889-6110
E-mail: jtu@cs.wustl.edu

March 23, 1990

A novel process algebra is presented; algebraic expressions specify delay-insensitive circuits in terms of voltage-level transitions on wires. The approach appears to have several advantages over traditional state-graph and production-rule based methods. The wealth of algebraic laws makes it possible to specify circuits concisely and facilitates the verification of designs. Individual components can be composed into circuits in which signals along internal wires are hidden from the environment.

0 Introduction

A circuit is connected to its environment by a number of wires. If the circuit functions correctly irrespective of the propagation delays in these wires, the circuit is called *delay-insensitive*. Delay-insensitive circuits are attractive because they can be designed in a modular way; indeed no timing constraints have to be satisfied in connecting such circuits together. As a result of the latest Turing Award Lecture [14], the design of delay-insensitive circuits has drawn renewed interest.

The design of delay-insensitive circuits is made difficult by the need to consider situations in which a signal (voltage-level transition) has been transmitted at one end of a wire but has not yet been received at the other end. The algebraic notation presented in this paper may be helpful in the following ways:

1. The functional behavior of primitive delay-insensitive components can be captured by algebraic expressions.

2. All possible behaviors of the circuit that results when such components are connected together can be determined by symbolic manipulation.

3. The algebra facilitates the precise specification of the circuit that the designer has to build, including obligations to be met by the environment.

4. The algebra supports verification of the design against the specification.

The algebra is based upon Hoare's CSP notation [8]. It adapts the theory of asynchronous processes [9, 10] to the special case of delay-insensitive circuits. The possibilities of *transmission interference* and *computation interference*, characterized by Udding [15, 16], are faithfully modeled in the algebra; the designer is able to reason about these errors, and so avoid them. Underpinning the algebra is a denotational semantics similar to those given in [2, 10]; the semantics is compatible with the failures-divergences model of CSP [1, 8].

Our approach complements that taken by Martin [11, 12] to the design of delay-insensitive circuits. Martin's approach, however, is more general in that he makes use of components that are not delay-insensitive, namely his *isochronic forks*. We have discovered that it is possible to understand many of Martin's circuits by treating the isochronic fork, together with one of the gates to which it connects, as a single primitive delay-insensitive component. (Ebergen [5], on the other hand, has investigated how components that are sensitive to delay can be synchronized to form delay-insensitive circuits.)

The remainder of this paper provides a step-by-step introduction to the algebra. We also prove, as a case study, some of Martin's nontrivial circuit designs to be algebraically correct. Similar verifications have been done by Dill [4, 3]. His verifications, however, are performed at the semantics level rather than at the syntactic/algebraic level. Algebraic calculations are arduous but humanly feasible, as well as mechanizable, and seem to avoid a state explosion.

1 Basic Notions and Operators

A process is a mathematical model at a certain level of abstraction of the way in which a delay-insensitive circuit interacts with its environment. Typical processes are P and Q. A circuit receives signals from its environment on its input wires and sends signals to its environment on its output wires. Thus with each process are associated an alphabet of input wires and an alphabet of output wires. These alphabets are finite and disjoint. Typical names for input wires are a and b; typical names for output wires are c and d. The time taken by a signal to traverse a wire is indeterminate.

In the remainder of this section and section 2, we consider processes with a particular alphabet I of input wires and a particular alphabet O of output wires.

The process P is considered to be "just as good" as the process Q ($P \subseteq Q$) if no environment, which is simply another process, can when interacting with P determine that it is not interacting with Q. (This is the refinement ordering of CSP [1, 8], also known as the *must* ordering [7].) Two processes are considered to be equal when they are just as good as each other.

The refinement ordering is intimately connected with nondeterministic choice. The process $P \sqcap Q$ is allowed to behave either as P or as Q. It reflects the designer's freedom to implement such a process by either P or Q. (Thus $\sqcap$ is obviously commutative, associative and idempotent.) Now $P \sqcap Q = Q$ exactly when $P \subseteq Q$.

A wire cannot accommodate two signals at the same time; they might interfere with one another in an undesirable way. This and any other error are modeled by the process $\bot$ (Bottom or Chaos). The environment must ensure that a process never gets into such a state. The process $\bot$ is considered to be so undesirable that any other process must be an improvement on it:

Law 0. $P \subseteq \bot$

It follows that $\sqcap$ has $\bot$ as a null element.

We shall mostly be concerned with recursively-defined processes. The meaning of the recursion $\mu X.F(X)$ is the least fixpoint of F. Its successive approximations are $\bot$, $F(\bot)$, $F(F(\bot))$, *etc.* All the operators that we shall use to define processes are continuous (and therefore monotonic); all except for recursion are distributive (with respect to nondeterministic choice).

In earlier approaches to an algebra for delay-insensitive circuits, *cf.* [15, 2, 5], a particular input signal is allowed only when this is explicitly indicated, and otherwise is assumed to lead to interference. This is in contrast with the algebra presented here: an input need not result in interference even though the possibility of such an input has not been made explicit in the algebraic expression. This follows the approach taken in [10] and is more convenient in algebraic manipulation, even though at first it may appear less natural.

Thus we write $a?;P$ to denote a process that must wait for a signal to arrive on $a \in I$ before it can behave like P. It is quite permissible for the environment to send a signal along any other input wire b. Such a signal is ignored at least until a signal is sent along a (or a second signal is sent along b causing interference).

A process that waits for input on a and then for input on b before being able to do anything is actually just waiting for inputs on both a and b, their order being immaterial:

Law 1. $a?;b?;P = b?;a?;P$

Complementary to input-prefixing $a?;P$ is output-prefixing $c!;P$, where $c \in O$. This is a process that outputs on c and then behaves like P. The environment may send a signal on any input wire even before it receives the signal on c; whether or not it can do so safely depends on P.

Two outputs by a process on the same wire, one after the other, is unsafe because of the danger of the two signals interfering with one another before they reach the environment. Also, since any output of a process may arrive at the environment an arbitrary time later, the order in which outputs are sent is immaterial. Therefore, we have the following two laws:

Law 2. $c!;c!;P = \bot$

Law 3. $c!;d!;P = d!;c!;P$

Example 0 Law 2. allows us to prove that $c!;\bot = \bot$. This should not be surprising: the process $c!;\bot$ behaves like $\bot$ after it has output on c; a wholly undesirable state results even before the output has reached the environment.

$$
\begin{aligned}
& c!;\bot \\
= \quad & \{ \text{ Law 2. } \} \\
& c!;c!;c!;\bot \\
= \quad & \{ \text{ Law 2. } \} \\
& \bot
\end{aligned}
$$

■

Finally, as in CSP, prefixing is distributive (with respect to nondeterministic choice). For both input and output-prefixing we have the law

Law 4. $x;(P \sqcap Q) = (x;P) \sqcap (x;Q)$

Example 1 We are now in a position to specify a number of elementary delay-insensitive components, *viz.* the Wire, the Fork, and the C-element.

Consider a circuit with one input wire a and one output wire c. In response to each signal on a, the circuit should produce a signal on c. The precise behavior of this circuit is given by the following algebraic expression:

$$\mu X.\ a?;c!;X$$

which we shall refer to as the process W because it can be readily implemented by a wire. Now unfolding the recursion, we have that $W = a?;c!;W$. As in CSP, this equation itself uniquely defines W because its right hand side is guarded.

Next consider the process, with one input wire a and two output wires c and d, defined by the equation $F = a?;c!;d!;F$. This models the behavior of a fork.

Finally, the Muller C-element repeatedly waits for inputs on wires a and b before outputting on c. It is defined by $C = a?;b?;c!;C$.

When we introduce the *after* operator in the next section, it will become clear that the above expressions do indeed correctly specify the components. ■

A more general form of input-prefixing is input-guarded choice. Such a choice allows a process to take different actions depending upon the input received. The choice is made between a number of alternatives of the form $a? \rightarrow P$. For S a finite set of alternatives, the guarded choice $[S]$ selects one of them. An alternative $a? \rightarrow P$ can be selected only if a signal has been received on a. The choice cannot be postponed indefinitely once one or more alternatives become selectable.

Choice with only one alternative is no real choice at all:

Law 5. $[a? \rightarrow P] = a?; P$

The environment cannot safely send a second signal along an input wire until the first signal has been acknowledged. Thus the result of sending two signals on a to the process $a?; a?; P$ is $\perp$ rather than P. The process is as useless as a choice with no alternative:

Law 6. $a?; a?; P = a?; [\,] = [\,]$

If two alternatives are guarded on a, then either may be chosen after input has been supplied on a. Indeed, the designer has the freedom to implement only one of the two. This is captured in the following law, where the symbol $\Box$ separates the various alternatives:

Law 7.
$$\begin{aligned} & [a? \rightarrow P \;\Box\; a? \rightarrow Q \;\Box\; S] \\ = \;& [a? \rightarrow (P \sqcap Q) \;\Box\; S] \\ = \;& [a? \rightarrow P \;\Box\; S] \sqcap [a? \rightarrow Q \;\Box\; S] \end{aligned}$$

Example 2 With the above law we can prove the following absorption theorem. An alternative guarded on a is absorbed by $a? \rightarrow \perp$:

$$\begin{aligned} & [a? \rightarrow \perp \;\Box\; a? \rightarrow P \;\Box\; S] \\ = \;& \{ \text{ combining alternatives using Law 7. } \} \\ & [a? \rightarrow (\perp \sqcap P) \;\Box\; S] \\ = \;& \{ \perp \text{ is the null element of } \sqcap \} \\ & [a? \rightarrow \perp \;\Box\; S] \end{aligned}$$

■

Until a process acknowledges receipt of an input signal, a second signal on the same wire can result in interference. So, for S_0 and S_1 sets of alternatives, we have

Law 8. $[a? \rightarrow [S_0] \;\square\; S_1] = [a? \rightarrow [a? \rightarrow \bot \;\square\; S_0] \;\square\; S_1]$

Example 3 As a matter of fact, we can replace $\bot$ in Law 8. by any process P.

$$
\begin{array}{ll}
 & [a? \rightarrow [a? \rightarrow \bot \;\square\; S_0] \;\square\; S_1] \\
= & \quad \{ \text{ absorption law derived in Example 2 } \} \\
 & [a? \rightarrow [a? \rightarrow \bot \;\square\; a? \rightarrow P \;\square\; S_0] \;\square\; S_1] \\
= & \quad \{ \text{ Law 8. } \} \\
 & [a? \rightarrow [a? \rightarrow P \;\square\; S_0] \;\square\; S_1]
\end{array}
$$

■

Indeed, if it is unsafe for the environment to send a signal along a particular input wire, it remains unsafe at least until an output has been received. Therefore, we also have the following absorption law.

Law 9. $[a? \rightarrow \bot \;\square\; b? \rightarrow [a? \rightarrow P \;\square\; S_0] \;\square\; S_1] \;=\; [a? \rightarrow \bot \;\square\; b? \rightarrow [S_0] \;\square\; S_1]$

Example 4 With the input-guarded choice we can model somewhat more interesting delay-insensitive components, such as the Merge, the Selector and the Decision-Wait element.

The Merge is a circuit with two input wires a and b and one output wire c. In response to a signal on either a or b, it will output on c:

$$M = [a? \rightarrow c!; M \;\square\; b? \rightarrow c!; M]$$

We shall discover, in the next section, that this definition implies that it is unsafe for the environment to supply input on both a and b before receiving an output on c.

The Selector is a circuit with one input wire a and two output wires c and d. Upon reception of an input it outputs on one of the two wires:

$$S = [a? \rightarrow c!; S \ \Box \ a? \rightarrow d!; S]$$

Actually, in this case there is no need to use guarded choice. By Laws 5. and 7., an equivalent formulation is $S = a?; ((c!; S) \sqcap (d!; S))$.

Finally, we can define the Decision-Wait element (2×1 in this case). It expects one input change in its row and one input change in its column. It produces as output the single entry which is indicated by the two changing inputs – there are two entries in this case:

$$\begin{aligned} DW \ = \ & [r0? \rightarrow [r1? \rightarrow \bot \ \Box \ c? \rightarrow e0!; DW] \\ & \Box \ \ r1? \rightarrow [r0? \rightarrow \bot \ \Box \ c? \rightarrow e1!; DW] \ ] \end{aligned}$$

(A C-element can be viewed as a 1×1 Decision-Wait element.) ■

2 More Advanced Constructs

In the last section we provided enough operators to allow us to specify many interesting delay-insensitive components from which larger circuits might be constructed. To better understand these specifications we need to be able to determine how a circuit will behave after some signals have been exchanged with its environment. Before we can do this, it turns out that we need a more general form of guarded choice which allows for *skip* guards as well as input guards.

Recall that a guarded choice $[S]$ consists of a set S of alternatives of the form $a? \rightarrow P$. We shall now allow S to include also alternatives of the form $skip \rightarrow P$. Such an alternative can be selected whether or not any input is supplied. (As in CSP, if no input is supplied, it must eventually be selected.)

The laws of the last section remain valid, but in addition we have several laws involving *skip* guards. (These are also laws in occam [13].) As before, a choice with one alternative is no real choice at all:

Law 10. $[skip \rightarrow P] = P$

Nondeterministic choice can be regarded as a special case of guarded choice:

Law 11. $[skip \rightarrow P \ \Box \ skip \rightarrow Q] = P \sqcap Q$

The selection of a *skip*-guarded alternative is an internal (unobservable) action of a process. This gives rise to the following three laws. In the first, a nondeterministic choice arises after input has been supplied on a because the signal may arrive before the selection of a *skip*-guarded alternative:

Law 12. $[a? \rightarrow P \ \Box \ skip \rightarrow [a? \rightarrow Q \ \Box \ S_0] \ \Box \ S_1]$
$= [skip \rightarrow [a? \rightarrow (P \sqcap Q) \ \Box \ S_0] \ \Box \ S_1]$

The second law states that a nested *skip*-guarded alternative can be selected in preference to any other alternative:

Law 13. $[skip \rightarrow [skip \rightarrow P \ \Box \ S_0] \ \Box \ S_1] = [skip \rightarrow P \ \Box \ S_0 \ \Box \ S_1]$

The third law is a convexity property of guarded choice:

Law 14. $[skip \rightarrow [S_0] \ \Box \ skip \rightarrow [S_0 \ \Box \ S_1] \ \Box \ S_2] = [skip \rightarrow [S_0] \ \Box \ S_1 \ \Box \ S_2]$

Example 5 Two *skip*-guarded alternatives can be combined together:

$[skip \rightarrow P \ \Box \ skip \rightarrow Q \ \Box \ S]$

$=$ { nesting the *skip* guards using Law 13. }

$[skip \rightarrow [skip \rightarrow P \ \Box \ skip \rightarrow Q] \ \Box \ S]$

$=$ { $\sqcap$ as guarded choice, Law 11. }

$[skip \rightarrow (P \sqcap Q) \ \Box \ S]$

■

Example 6 With impunity we can extend the set of alternatives in a guarded choice with that guarded choice itself as a new *skip*-guarded alternative:

$[S]$

$=$ { one choice is no choice, Law 10. }

$[skip \rightarrow [S]\ ]$

$=$ { convexity, Law 14. }

$$[skip \rightarrow [S] \;\Box\; S]$$

■

Example 7 A *skip*-guarded alternative can always be chosen, and so no other alternative need be offered, *i.e.*, $P \subseteq [skip \rightarrow P \;\Box\; S]$.

$$\begin{array}{ll} & P \sqcap [skip \rightarrow P \;\Box\; S] \\ = & \quad \{ \sqcap \text{ as guarded choice, Law 11. } \} \\ & [skip \rightarrow P \;\Box\; skip \rightarrow [skip \rightarrow P \;\Box\; S]\,] \\ = & \quad \{ \text{ unnesting the } skip \text{ guards, Law 13. } \} \\ & [skip \rightarrow P \;\Box\; S] \end{array}$$

In particular, $[skip \rightarrow \bot \;\Box\; S] = \bot$ by Law 0. ■

Example 8 As another example of interference between two consecutive outputs on a wire, we have

$$\begin{array}{ll} & c!; [skip \rightarrow c!; P \;\Box\; S] \\ \supseteq & \quad \{ \text{ Example 7 and monotonicity of prefixing } \} \\ & c!; c!; P \\ = & \quad \{ \text{ interference between outputs, Law 2. } \} \\ & \bot \end{array}$$

which means that $c!; [skip \rightarrow c!; P \;\Box\; S] = \bot$, by Law 0. ■

We are now able to define how a process behaves after the environment has sent input to it or received output from it. We consider the after-input case first.

The process $P/a?$ behaves like P after its environment has sent it a signal on $a \in I$. Notice that this does not mean that P has received this input yet; the signal may still be on its way. Indeed it is impossible to tell whether P has received the signal until some acknowledging signal has been received from P.

The first two laws for after-input are concerned with undesirable behavior. A process which has entered an unsafe state remains unsafe. Also, sending two signals in a row on an input wire may cause interference and is therefore unsafe.

Law 15. $\bot/a? = \bot$

Law 16. $P/a?/a? = \bot$

The order in which signals are sent does not determine the order in which they are received, and so

Law 17. $P/a?/b? = P/b?/a?$

An output-prefixed process can do nothing but output, even when sent input:

Law 18. $(c!;P)/a? = c!;(P/a?)$

After-input (on a) distributes through the alternatives in a guarded choice, except for those alternatives guarded on a. Those become *skip*-guarded. Furthermore, an extra alternative is required to indicate that interference can result after a second input on a.

Law 19. $[S]/a? = [a? \rightarrow \bot \ \Box \ S']$,
where S' is formed by substituting for each alternative $A \in S$ the new alternative $A/a?$, defined by

$$(skip \rightarrow P)/a? = skip \rightarrow (P/a?)$$
$$(a? \rightarrow P)/a? = skip \rightarrow P$$
$$(b? \rightarrow P)/a? = b? \rightarrow (P/a?), \text{ for } b \neq a.$$

As a consequence of Laws 5. and 19., we have that

$$(a?;P)/a? = [a? \rightarrow \bot \ \Box \ skip \rightarrow P]$$

and

$$(b?;P)/a? = [a? \rightarrow \bot \ \Box \ b? \rightarrow (P/a?)].$$

Example 9 When $\bot$ is guarded on a, the environment must not supply input on a.

$$[a? \rightarrow \bot \ \Box \ S]/a?$$
$$= \quad \{ \text{ after through guarded choice, Law 19., } S' \text{ being some set of alternatives derived from } S \ \}$$

$[a? \to \bot \;\Box\; skip \to \bot \;\Box\; S']$

$=$ { unguarded $\bot$, Example 7 }

$\bot$

■

The following law allows us to expand the set of alternatives in a guarded choice to make the behavior after a particular input explicit:

Law 20. $[S] = [a? \to [S]/a? \;\Box\; S]$

Example 10 It follows that after-input is distributive:

$(P/a?) \sqcap (Q/a?)$

$=$ { $\sqcap$ as guarded choice, Law 11. }

$[skip \to (P/a?) \;\Box\; skip \to (Q/a?)]$

$=$ { adding an a-guarded alternative with Law 20. }

$[a? \to [skip \to (P/a?) \;\Box\; skip \to (Q/a?)]/a?$

$\Box\; skip \to (P/a?) \;\Box\; skip \to (Q/a?)]$

$=$ { interference between inputs using Laws 16. and 19. and Example 7 }

$[a? \to \bot \;\Box\; skip \to (P/a?) \;\Box\; skip \to (Q/a?)]$

$=$ { after through guarded choice, Law 19. }

$[skip \to P \;\Box\; skip \to Q]/a?$

$=$ { $\sqcap$ as guarded choice, Law 11. }

$(P \sqcap Q)/a?$

■

Example 11 Here is a case in which *skip* can be eliminated:

$[a? \to \bot \;\Box\; skip \to [S]]$

$=$ { adding an a-guarded alternative with Law 20. }

$[a? \rightarrow \bot \ \Box \ skip \rightarrow [a? \rightarrow ([S]/a?) \ \Box \ S]]$

$=$ { postponing alternative until after *skip*, Law 12. }

$[skip \rightarrow [a? \rightarrow (([S]/a?) \sqcap \bot) \ \Box \ S]]$

$=$ { one choice is no choice and $\bot$ is null element of $\sqcap$ }

$[a? \rightarrow \bot \ \Box \ S]$

■

Example 12 If $[S]/a? = \bot$, then $[a? \rightarrow \bot \ \Box \ S] = [S]$:

$[a? \rightarrow \bot \ \Box \ S]$

$=$ { $[S]/a? = \bot$ }

$[a? \rightarrow [S]/a? \ \Box \ S]$

$=$ { removing a-guarded alternative with Law 20. }

$[S]$

■

The following law is a useful generalization of Example 12:

Law 21. If $[S]/a? = \bot$, then $[a? \rightarrow P \ \Box \ S] = [S]$.

The process $P/c!$ behaves as P after the environment has received a signal on $c \in O$. It does not make sense to define after-output for just any output, but only for those that are actually possible.

Definition 0 We define $out(P)$, the set of wires on which P can output initially (and so for which after-output is defined), as follows:

$$out(\bot) = O$$

$$out(c!; P) = \begin{cases} O & \text{if } c \in out(P) \\ \{c\} \cup out(P) & \text{otherwise} \end{cases}$$

$out([\,]) = \emptyset$

$out([a? \rightarrow P \;\square\; S]) = out([S])$

$out([skip \rightarrow P \;\square\; S]) = out(P) \cup out([S])$

$out(\mu X.F(X)) = (\bigcap n : n \geq 0 : out(F^n(\bot)))$

■

It follows from this (by structural induction) that $out(P) \subseteq out(P/a?)$. Also note that another way to define $out(P)$ is that $c \in out(P)$ if and only if $c!; P = \bot$.

We can now examine the laws for after-output. In Laws 24.-27. it is understood that expressions involving after-output are well-defined, *i.e.*, $c \in out(d!; P)$, $c \in out([S])$, $c, d \in out(P)$ and $c \in out(P)$, respectively.

Law 22. $\bot/c! = \bot$

Law 23. $(c!; P)/c! = \begin{cases} \bot & \text{if } c \in out(P) \\ P & \text{otherwise} \end{cases}$

Law 24. For $c \neq d$, $(d!; P)/c! = \begin{cases} \bot & \text{if } d \in out(P) \\ d!; (P/c!) & \text{otherwise} \end{cases}$

Law 25. $[S]/c! = [S']$,
where S' consists of an alternative $skip \rightarrow (P/c!)$ for each alternative in S of the form $skip \rightarrow P$ for which $c \in out(P)$.

Law 26. $P/c!/d! = P/d!/c!$

Law 27. $P/c!/a? \subseteq P/a?/c!$

Law 27. may be understood as follows. It is assumed that P can output on c. Even when the environment receives the output on c after having supplied the input on a, P might have output on c before receiving that input. Thus the possible behaviors of $P/a?/c!$ include those of $P/c!/a?$. As an example, consider $P = [skip \rightarrow c!; [\,] \;\square\; a? \rightarrow \bot]$. Then, $P/c!/a? = a?; \bot$, but $P/a?/c! = \bot$.

Example 13 Now we can determine how components such as the C-element and the Merge behave after they have interacted with their environment. For the C-element we derive

$(a?;b?;c!;C)/a?$

$=$ { after through input-prefixing, corollary to Law 19. }

$[a? \rightarrow \bot \ \square \ skip \rightarrow b?;c!;C]$

$=$ { one choice is no choice and eliminating *skip* as in Example 11 }

$[a? \rightarrow \bot \ \square \ b? \rightarrow c!;C]$

Hence, a C-element that has been sent one input on a must not be sent another (until a signal on c has been received). Before it will output on c, however, it has to input on b. This is exactly how we want a C-element to behave after being sent a signal on a. We can now also compute $C/a?/b?$. A little calculation shows that the result is $[a? \rightarrow \bot \ \square \ b? \rightarrow \bot \ \square \ skip \rightarrow c!;C]$, as desired. Taking it one step further, we can show that $C = C/a?/b?/c!$.

A more interesting example of the possibility of interference is seen in the specification of Merge. Once a signal has been sent on either input wire, both input wires become unsafe to use. In this case we compute

$M/a?$

$=$ { definition of M }

$[a? \rightarrow c!;M \ \square \ b? \rightarrow c!;M]/a?$

$=$ { after through choice and prefixing, Laws 18. and 19. }

$[a? \rightarrow \bot \ \square \ skip \rightarrow c!;M \ \square \ b? \rightarrow c!;(M/a?)]$

$=$ { $M/a?$ is of the form $[skip \rightarrow c!;P \ \square \ S]$, Example 8 }

$[a? \rightarrow \bot \ \square \ b? \rightarrow \bot \ \square \ skip \rightarrow c!;M]$

This shows that signals on both a and b should be withheld until the Merge outputs on c. ■

3 Composition

In this section we define a parallel composition operator. With it we can determine the overall behavior of a circuit from the individual behavior of its components.

It is understood that if the output wire of one component has the same name as the input wire of another, then these wires are supposed to be joined together; any signals transmitted along such a connection are hidden from the environment. The parallel composition operator is fundamental to a hierarchical approach to circuit design. It permits an initial specification to be decomposed into a number of components operating in parallel, and each of these components can be designed independently of the rest.

The simplicity of the laws enjoyed by parallel composition is one of the main attractions of our algebra. Indeed, in [15] certain restrictions had to be placed on processes before their composition could even be considered; and in [2] the fixed-point definition of parallel composition was rather unwieldy.

Parallel composition is denoted by the infix binary operator $\|$. All the operators we have met so far do not affect the input and output alphabets of their operands; so, for example, in the nondeterministic choice $P \sqcap Q$, we insist that the input alphabet of P is the same as that of Q, and declare that it is the same as that of $P \sqcap Q$. In the parallel composition $P \parallel Q$, however, 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. (These rules prohibit fan-in and fan-out of wires; the explicit use of Merges and Forks is required.) The input alphabet of $P \parallel Q$ then consists of those input wires of each process P and Q which are not output wires of the other. Similarly, the output alphabet of $P \parallel Q$ consists of those output wires of each process which are not input wires of the other.

Parallel composition is commutative. It is also associative, provided we ensure that a wire named in the alphabets of any two processes being composed is not in the alphabets of a third process. If one process in a parallel composition is in an undesirable state, then the overall state is undesirable:

Law 28. $P \parallel \bot = \bot$

When an output-prefixed process $c!;P$ is composed with another process Q, the output is transmitted along c. Depending on whether or not c is in the input alphabet of Q, the signal on c is sent to Q or to the environment:

Law 29. $$(c!;P) \parallel Q = \begin{cases} P \parallel (Q/c?) & \text{if } c \text{ is in the input alphabet of } Q \\ c!;(P \parallel Q) & \text{otherwise} \end{cases}$$

It remains only to consider parallel composition of guarded choices. The following law specifies the alternatives in the resulting guarded choice.

Law 30. $[S_0] \parallel [S_1] = [S]$,

where S is formed from the alternatives in S_0 and S_1 in the following way. For each alternative in S_0 of the form $skip \rightarrow P$, we have $skip \rightarrow (P \parallel [S_1])$ in S. For each alternative in S_0 of the form $a? \rightarrow P$ with a not in the output alphabet of $[S_1]$, we have $a? \rightarrow (P \parallel [S_1])$ in S. The alternatives in S_1 contribute to the alternatives in S in a similar way.

Example 14 If one component is able to send a signal that it is unsafe for the other to receive, then their parallel composition is $\perp$.

$$
\begin{aligned}
& (a!; P) \parallel [a? \rightarrow \perp \ \Box \ S] \\
= \quad & \{ \text{ internal communication on } a, \text{ Law 29. } \} \\
& P \parallel [a? \rightarrow \perp \ \Box \ S]/a? \\
= \quad & \{ \text{ Example 9 and } \perp \text{ null element of parallel composition, Law 28. } \} \\
& \perp
\end{aligned}
$$

■

Example 15 We compute a number of simple compositions in this example. Although the resulting behaviors are well-known, it has never previously been possible to give a straightforward algebraic derivation.

Consider first connecting two wires W_0 and W_1 together. Let $W_0 = a?; b!; W_0$ and $W_1 = b?; c!; W_1$. Then, in their parallel composition, signals on b are hidden from the environment.

$$
\begin{aligned}
& W_0 \parallel W_1 \\
= \quad & \{ \text{ definitions of } W_0 \text{ and } W_1 \ \} \\
& (a?; b!; W_0) \parallel (b?; c!; W_1) \\
= \quad & \{ \text{ one choice is no choice and parallel composition through guarded choice, Law 30., using that } b \text{ is internal } \} \\
& a?; ((b!; W_0) \parallel (b?; c!; W_1)) \\
= \quad & \{ \text{ internal communication on } b, \text{ Law 29. } \} \\
& a?; (W_0 \parallel (b?; c!; W_1)/b?)
\end{aligned}
$$

$=$ { after through prefixing }

$a?;(W_0 \parallel [b? \to \bot \;\Box\; skip \to c!;W_1])$

$=$ { substituting for W_0 and applying Law 30., parallel composition through guarded choice, using that b is internal }

$a?;\ [\ \ a? \to ((b!;W_0) \parallel [b? \to \bot \;\Box\; skip \to c!;W_1])$
$\Box\ skip \to (W_0 \parallel (c!;W_1))\ ]$

$=$ { one choice is no choice and absorption as in Example 3 }

$a?;[skip \to (W_0 \parallel (c!;W_1))]$

$=$ { one choice is no choice and external communication with Law 29. }

$a?;c!;(W_0 \parallel W_1)$

Since this recursion is guarded, we conclude that $W_0 \parallel W_1 = W$.

A more interesting example is the composition of a "one-hot" C-element and a Fork in the following way. The C-element is specified by $C = a?;b?;c!;C$ and the Fork by $F = c?;a!;d!;F$. This is a circuit involving feedback.

$C/a? \parallel F$

$=$ { Example 13 and definition of F }

$[a? \to \bot \;\Box\; b? \to c!;C] \parallel (c?;a!;d!;F)$

$=$ { one choice is no choice and parallel composition through guarded choice, using that a and c are internal }

$b?;((c!;C) \parallel (c?;a!;d!;F))$

$=$ { internal communication on c }

$b?;(C \parallel (c?;a!;d!;F)/c?)$

$=$ { definition of C and after through prefixing }

$b?;((a?;b?;c!;C) \parallel [c? \to \bot \;\Box\; skip \to a!;d!;F])$

$=$ { one choice is no choice, parallel composition through guarded choice, using that a and c are internal, and definition of C }

$b?;[skip \to (C \parallel (a!;d!;F))]$

$=$ { one choice is no choice, internal communication on a and external communication on d }

$b?; d!; (C/a? \parallel F)$

By uniqueness of guarded recursion, this combination of C-element and Fork behaves just like a wire. Although the Fork signals on a and d "in parallel", this did not lead to a doubling of the number of states which we had to analyze. We could deal with a entirely before d was pulled out of the parallel composition. This technique can be more generally applied and that is why these algebraic manipulations do not lead to a state explosion. ■

4 A Small Case Study

In [12] Martin presents a number of designs of circuits that are delay-insensitive but for their use of Isochronic Forks. An Isochronic Fork has one input wire and two output wires and operates in a way similar to the Fork of Example 1, except that

> the difference in delays between the two branches of the fork is shorter than the delays in the operators to which the fork is an input.

Thus the environment needs to register the arrival of only one output signal before it can safely send another signal to the fork. The Isochronic Fork clearly does not connect components in a delay-insensitive way. Fortunately, one output of an Isochronic Fork is usually fed into an AND-element and this combination as a whole, which we call an Isochronic AND-element (Figure 0), is insensitive to delay. Treating the Isochronic Fork and an AND-element to which it connects as one delay-insensitive component enables us to verify many of Martin's designs. (This treatment suffices for the verification of many designs but not all, because the property of Isochronic Forks stated above is actually slightly stronger than the property that we choose to model.) In particular, our algebra clarifies the relationship between his Q-element and D-element, and we show in this section that a single proof suffices for the verification of his designs for both these components. First, we give the specification of an Isochronic And-element, which we deduce from the assumed behavior of an Isochronic Fork.

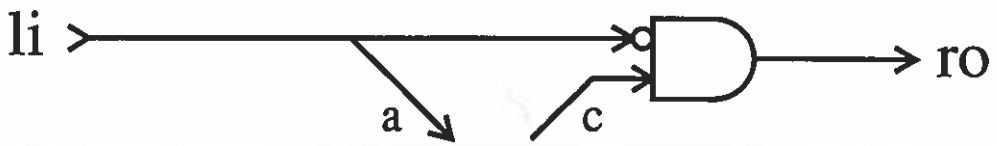

Figure 0: The delay-insensitive Isochronic And-element. (The internal connection of fork to AND-element is sensitive to delay, but the external connections are all delay-insensitive.)

As usual, we assume that all wires connecting a component to its environment are initially low. Signals on a wire between a component and its environment are either up-going or down-going transitions. The problem with the AND-element in delay-insensitive circuit design is that once both of its input wires are high, we can never allow both of them to go low again. The AND-element would output only once and, hence, acknowledge only one of the two down-going inputs. There is no way that the environment could tell that the signal on the other input wire had been received. When the AND-element is combined with an Isochronic Fork, however, it is possible to allow both its input wires to go low again. One branch of the fork connects to one input wire of the AND-element. Any signal that reaches the AND-element on that wire is effectively acknowledged by the corresponding signal on the other branch of the fork. We treat the Isochronic AND-element as a primitive delay-insensitive component and do not model the Isochronic Fork on its own.

The behavior of an Isochronic AND-element (IND for short) can be specified by four mutually-recursive equations. Each defines a quiescent state of the component, *i.e.*, a state in which nothing will be output until further input is supplied. The processes IND$^\circ$ and IND$^\bullet$ may be thought of as "one-hot" Isochronic AND-elements, one input wire being high in each case. IOR may be thought of as an Isochronic OR-element.

Definition 1 An Isochronic AND-element is defined as follows, where input to the Isochronic Fork is on wire li, output from the Isochronic Fork to the environment is on wire a, input from the environment to the AND-element is on wire c and output from the AND-element to the environment is on wire ro.

$$
\begin{array}{lcl}
\mathrm{IND} & = & [li? \rightarrow a!; \mathrm{IND}^{\bullet} \;\Box\; c? \rightarrow \mathrm{IND}^{\circ}] \\
\mathrm{IND}^{\bullet} & = & [li? \rightarrow a!; \mathrm{IND} \;\Box\; c? \rightarrow ro!; \mathrm{IOR}] \\
\mathrm{IND}^{\circ} & = & [li? \rightarrow a!; ro!; \mathrm{IOR} \;\Box\; c? \rightarrow \bot] \\
\mathrm{IOR} & = & [li? \rightarrow a!; ro!; \mathrm{IND}^{\circ} \;\Box\; c? \rightarrow ro!; \mathrm{IND}^{\bullet}]
\end{array}
$$

■

In checking that this definition is reasonable, one discovers, for example, that $\mathrm{IND}/li?/a!/c?/li? = \bot$. (One input wire to an AND-element going low while the other goes high may cause a glitch.)

The following lemmas will be useful later. In some of the proofs we abbreviate certain processes to P and Q and certain sets of alternatives to S_0 and S_1 when their exact expression is no longer of interest.

Lemma 0 $(ro!; \mathrm{IOR})/li? = \bot$.

Proof

$$
\begin{array}{ll}
 & (ro!; \mathrm{IOR})/li? \\
= & \quad \{ \text{ After through prefixing and guarded choice } \} \\
 & ro!; [li? \rightarrow \bot \;\Box\; skip \rightarrow a!; ro!; \mathrm{IND}^{\circ} \;\Box\; c? \rightarrow (ro!; \mathrm{IND}^{\bullet})/li?] \\
= & \quad \{ \text{ Example 8, using that } a!; ro!; P = ro!; a!; P \} \\
 & \bot
\end{array}
$$

■

Lemma 1 $(ro!; \mathrm{IOR})/c? = \bot$.

Proof

$$
\begin{array}{ll}
 & (ro!; \mathrm{IOR})/c? \\
= & \quad \{ \text{ After through prefixing and guarded choice } \} \\
 & ro!; [li? \rightarrow (a!; ro!; \mathrm{IND}^{\circ})/c? \;\Box\; c? \rightarrow \bot \;\Box\; skip \rightarrow ro!; \mathrm{IND}^{\bullet}] \\
= & \quad \{ \text{ Example 8 } \}
\end{array}
$$

$\bot$

■

Lemma 2 $\mathrm{IND}^\bullet/c? = ro!; \mathrm{IOR}$.

Proof

$\mathrm{IND}^\bullet/c?$

$=$ { After through guarded choice }

$[li? \rightarrow (a!; \mathrm{IND})/c? \;\Box\; c? \rightarrow \bot \;\Box\; skip \rightarrow ro!; \mathrm{IOR}]$

$=$ { on account of Lemma 1 and Example 7 we have $[li? \rightarrow P \;\Box\; skip \rightarrow ro!; IOR]/c? = \bot$. Apply Example 12 }

$[li? \rightarrow (a!; \mathrm{IND})/c? \;\Box\; skip \rightarrow ro!; \mathrm{IOR}]$

$=$ { Law 21., using Lemma 0 }

$[skip \rightarrow ro!; \mathrm{IOR}]$

$=$ { one choice is no choice }

$ro!; \mathrm{IOR}$

■

Lemma 3 $\mathrm{IND}/c? = \mathrm{IND}^\circ$.

Proof

$\mathrm{IND}/c?$

$=$ { After through guarded choice and prefixing }

$[li? \rightarrow a!; (\mathrm{IND}^\bullet/c?) \;\Box\; c? \rightarrow \bot \;\Box\; skip \rightarrow \mathrm{IND}^\circ]$

$=$ { Lemma 2 and Example 6 }

IND°

■

The D and Q-Elements

We are now ready to consider Martin's D and Q-elements [12, pp. 30-33]. Each provides an interface between a left environment and a right environment: they communicate with the left environment by inputting on li and outputting on lo, and with the right environment by inputting on ri and outputting on ro. Both use a four-cycle signaling scheme to implement a one-place buffer. The left environment fills the buffer by signaling on li (which is acknowledged on lo), and this cycle has to be repeated (return-to-zero); the signal ro empties the buffer (which the right environment acknowledges on ri), and this cycle also has to be repeated. Four mutually-recursive equations serve to define both the D and Q-elements.

Definition 2

$$\begin{aligned} D &= [li? \rightarrow lo!; Q \;\square\; ri? \rightarrow \bot] \\ Q &= [li? \rightarrow ro!; \tilde{D} \;\square\; ri? \rightarrow \bot] \\ \tilde{D} &= [ri? \rightarrow ro!; \tilde{Q} \;\square\; li? \rightarrow \bot] \\ \tilde{Q} &= [ri? \rightarrow lo!; D \;\square\; li? \rightarrow \bot] \end{aligned}$$

(Note that $\tilde{D}$ and $\tilde{Q}$ are just D and Q, respectively, with left and right switched over. Notice also that in any state exactly one of the two inputs is expected.) ■

Martin's designs for each element involve essentially two IND's, a C-element, and a Fork. These can be verified as follows.

Straightforward calculation shows that the C-element $C = a?; b?; x!; C$ composed with the Fork $F = x?; c!; d!; F$ yields the following C-element with two outputs

$$\mathrm{CF} = a?; b?; c!; d!; \mathrm{CF},$$

which we will use in the verification. As a matter of fact, we start out with $\mathrm{CF}/b?$ the specification of which is easily computed to be

$$\mathrm{CF}/b? = [b? \rightarrow \bot \;\square\; a? \rightarrow c!; d!; \mathrm{CF}].$$

It is also easy to see that

$$\mathrm{CF}/b?/a? = [a? \rightarrow \bot \;\square\; b? \rightarrow \bot \;\square\; skip \rightarrow c!; d!; \mathrm{CF}].$$

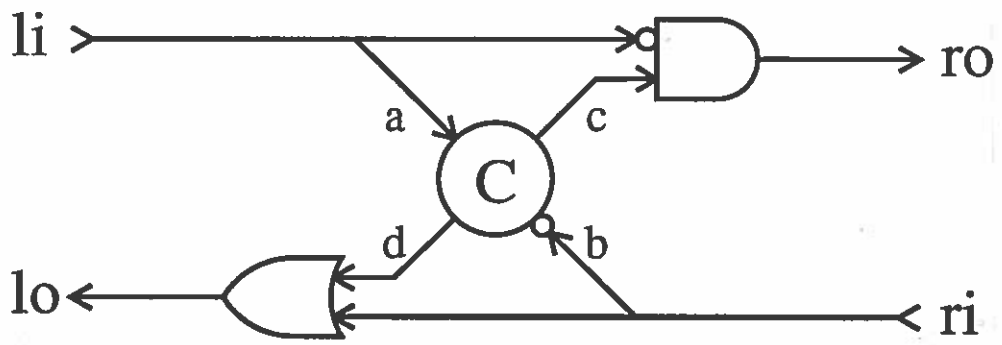

Figure 1: The composition of an IND•, an $\overline{\text{IOR}}$, a C-element, and a Fork

Martin's design for a D-element is depicted in Figure 1. It uses an IND• and an IOR. We need to rename the wires of the latter, so that li becomes ri, ro becomes lo, a becomes b and c becomes d; we shall denote such a change by an overbar. The specification of IND• and its derivatives are given in Definition 1. The specification of $\overline{\text{IOR}}$ and its derivatives are, according to the above renaming,

$$\begin{array}{lcl}
\overline{\text{IND}} & = & [ri? \rightarrow b!; \overline{\text{IND}}^{\bullet} \;\Box\; d? \rightarrow \overline{\text{IND}}^{\circ}] \\
\overline{\text{IND}}^{\bullet} & = & [ri? \rightarrow b!; \overline{\text{IND}} \;\Box\; d? \rightarrow lo!; \overline{\text{IOR}}] \\
\overline{\text{IND}}^{\circ} & = & [ri? \rightarrow b!; lo!; \overline{\text{IOR}} \;\Box\; d? \rightarrow \bot] \\
\overline{\text{IOR}} & = & [ri? \rightarrow b!; lo!; \overline{\text{IND}}^{\circ} \;\Box\; d? \rightarrow lo!; \overline{\text{IND}}^{\bullet}]
\end{array}$$

We now compute $\text{CF}/b? \parallel \text{IND}^{\bullet} \parallel \overline{\text{IOR}}$ and show that it implements the D-element.

$\text{CF}/b? \parallel \text{IND}^{\bullet} \parallel \overline{\text{IOR}}$

$=$ { parallel composition through guarded choice, using the specifications of $\text{CF}/b?$, $\text{IND}^{\bullet}$, and $\overline{\text{IOR}}$, and the fact that a, b, c and d are internal wires }

$[li? \rightarrow (\text{CF}/b? \parallel (a!; \text{IND}) \parallel \overline{\text{IOR}}) \;\Box\; ri? \rightarrow (\text{CF}/b? \parallel \text{IND}^{\bullet} \parallel (b!; lo!; \text{IND}^{\circ}))]$

$=$ { Example 14, using that $\text{CF}/b?$ is of the form $[b? \rightarrow \bot \;\Box\; S]$ }

$[li? \rightarrow (\text{CF}/b? \parallel (a!; \text{IND}) \parallel \overline{\text{IOR}}) \;\Box\; ri? \rightarrow \bot]$

$=$ { internal communication on a }

$$[li? \rightarrow (\mathrm{CF}/b?/a? \parallel \mathrm{IND} \parallel \overline{\mathrm{IOR}}) \;\Box\; ri? \rightarrow \bot]$$

$=$ { parallel composition through guarded choice, because a, b, c, and d are internal }

$$\begin{array}{l}[\quad li? \rightarrow [skip \rightarrow ((c!; d!; \mathrm{CF}) \parallel \mathrm{IND} \parallel \overline{\mathrm{IOR}}) \;\Box\; li? \rightarrow P \;\Box\; ri? \rightarrow Q] \\ \Box \quad ri? \rightarrow \bot]\end{array}$$

$=$ { Law 9, Example 3, and one choice is no choice }

$$[li? \rightarrow ((c!; d!; \mathrm{CF}) \parallel \mathrm{IND} \parallel \overline{\mathrm{IOR}}) \;\Box\; ri? \rightarrow \bot]$$

$=$ { internal communication on c and d and Lemma 3 }

$$[li? \rightarrow (\mathrm{CF} \parallel \mathrm{IND}^{\circ} \parallel \overline{\mathrm{IOR}}/d?) \;\Box\; ri? \rightarrow \bot]$$

$=$ { after through guarded choice }

$$\begin{array}{l}[\quad li? \rightarrow (\mathrm{CF} \parallel \mathrm{IND}^{\circ} \parallel [ri? \rightarrow P \;\Box\; d? \rightarrow \bot \;\Box\; skip \rightarrow lo!; \overline{\mathrm{IND}^{\bullet}}]) \\ \Box \quad ri? \rightarrow \bot]\end{array}$$

$=$ { parallel composition through guarded choice, because a, b, c, and d are internal }

$$\begin{array}{l}[\quad li? \rightarrow [li? \rightarrow Q \;\Box\; ri? \rightarrow R \;\Box\; skip \rightarrow (\mathrm{CF} \parallel \mathrm{IND}^{\circ} \parallel (lo!; \overline{\mathrm{IND}^{\bullet}}))] \\ \Box \quad ri? \rightarrow \bot]\end{array}$$

$=$ { Law 9, Example 3 and one choice is no choice }

$$[li? \rightarrow (\mathrm{CF} \parallel \mathrm{IND}^{\circ} \parallel (lo!; \overline{\mathrm{IND}^{\bullet}})) \;\Box\; ri? \rightarrow \bot]$$

$=$ { external communication on lo }

$$[li? \rightarrow lo!; (\mathrm{CF} \parallel \mathrm{IND}^{\circ} \parallel \overline{\mathrm{IND}^{\bullet}}) \;\Box\; ri? \rightarrow \bot]$$

Now if we look at the definition of the D-element we see that the beginning is there. Left to prove is that Q is implemented by $\mathrm{CF} \parallel \mathrm{IND}^{\circ} \parallel \overline{\mathrm{IND}^{\bullet}}$, which is Martin's design for a Q-element! We derive

$$\mathrm{CF} \parallel \mathrm{IND}^{\circ} \parallel \overline{\mathrm{IND}^{\bullet}}$$

$=$ { parallel composition through guarded choice, because a, b, c, and d are internal }

$$[li? \rightarrow (\mathrm{CF} \parallel (a!; ro!; \mathrm{IOR}) \parallel \overline{\mathrm{IND}^{\bullet}}) \;\Box\; ri? \rightarrow (\mathrm{CF} \parallel \mathrm{IND}^{\circ} \parallel (b!; \overline{\mathrm{IND}}))]$$

$\subseteq$ { guarded choice is monotonic with respect to refinement, internal communication on a and external communication on ro }

$$[li? \rightarrow ro!; (\mathrm{CF}/a? \parallel \mathrm{IOR} \parallel \overline{\mathrm{IND}^{\bullet}}) \;\Box\; ri? \rightarrow \bot]$$

(Note that we have indeed weakened the specification: the implementation is capable of more than that required by the specification.)

Now we use symmetry. We interchange a and b, c and d, li and ri, and lo and ro. For the two Isochronic elements this means that the top and bottom elements are interchanged. The C-element, however, is unaffected on account of Laws 1. and 3. Hence, we have also proved that

$$\mathrm{CF}/a? \parallel \overline{\mathrm{IND}^{\bullet}} \parallel \mathrm{IOR} \;=\; [ri? \rightarrow ro!; (\mathrm{CF} \parallel \overline{\mathrm{IND}^{\circ}} \parallel \mathrm{IND}^{\bullet}) \;\Box\; li? \rightarrow \bot]$$

and

$$\mathrm{CF} \parallel \overline{\mathrm{IND}^{\circ}} \parallel \mathrm{IND}^{\bullet} \;\subseteq\; [ri? \rightarrow lo!; (\mathrm{CF}/b? \parallel \overline{\mathrm{IOR}} \parallel \mathrm{IND}^{\bullet}) \;\Box\; li? \rightarrow \bot]$$

Using the commutativity and associativity of parallel composition, the unique solution of guarded recursions and the monotonicity of recursion, we conclude that

$$\mathrm{CF}/b? \parallel \mathrm{IND}^{\bullet} \parallel \overline{\mathrm{IOR}} \;\subseteq\; D$$

$$\mathrm{CF} \parallel \mathrm{IND}^{\circ} \parallel \overline{\mathrm{IND}^{\bullet}} \;\subseteq\; Q$$

That is, through algebraic manipulation we have verified Martin's designs for the D and Q-elements.

5 Conclusion

An algebraic approach has been taken to the specification and verification of delay-insensitive circuits. It has not been necessary to express explicitly all the states that such a circuit can enter; instead the possibility of them arising can be deduced using algebraic laws. This has led to concise specifications and short proofs. Another simplifying factor has been that, following [15], we do not distinguish between high and low-going transitions; this exposes many symmetries that would

not otherwise be apparent. The main advantage of our approach is the ease with which we can compute the parallel composition of components. We have worked through many examples in which we used algebraic laws either to prove further laws or to investigate the behavior of specified circuits. As a case study, we verified some of Martin's designs, bringing to light interesting facts about his Isochronic Forks, D-elements and Q-elements.

We have postulated a large number of laws in this paper. It is possible to give a semantics for the algebraic expressions, compatible with the failures-divergence model of CSP [1, 8], so as to prove the laws correct with respect to that semantics. This will establish the soundness of the algebra. A topic of future research is that of the completeness of the algebra. This means that any two expressions with the same meaning can algebraically be transformed into one another. Finally, we would like to answer the question what processes with a delay-insensitive semantics, cf. [2, 10], can be denoted by a term in the algebra. Since we allow processes to be defined recursively, we have left the realm of regular languages. However, what exactly can and cannot be expressed in the algebra requires further study.

In the case study of the previous section and in the earlier examples the algebra has merely been used for post hoc verification. Given a specification and a proposed design we have verified the correctness of the design. We have chosen to show how expressions can be manipulated in the algebra rather than to show how (complicated) expressions can be written as the parallel composition of (simpler) expressions. Preliminary research indicates, however, that these expressions do indeed suggest ways in which they can be decomposed. For example, the specifications of the D-element and the Q-element suggest a decomposition into a couple of Toggles and Merges. This has also been observed by Ebergen [6]. We believe that the algebra is a powerful tool for both design and verification, although only the latter has been addressed in this paper.

Acknowledgements

We are most grateful to Tony Hoare and Tom Verhoeff for their interest and encouragement. The hospitality of the Department of Computer Science at Washington University helped make it possible for us to collaborate over this research. The work was partially funded by the Science and Engineering Research Council of Great Britain and the ESPRIT Basic Research Action CONCUR.

References

[1] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating sequential processes. In G. Winskel, editor, *Proceedings of the NSF-SERC Seminar on Concurrency*, number 197 in Lecture Notes in Computer Science, pages 281–305. Springer-Verlag, 1985.

[2] W. Chen, J. T. Udding, and T. Verhoeff. Networks of communicating processes and their (de)-composition. In J. L. A. van de Snepscheut, editor, *The Mathematics of Program Construction*, number 375 in Lecture Notes in Computer Science, pages 174–196. Springer-Verlag, 1989.

[3] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* PhD thesis, C.S. Dept., Carnegie Mellon Univ., Pittsburgh, PA, Feb. 1988.

[4] D. L. Dill and E. M. Clarke. Automatic verification of asynchronous circuits using temporal logic. In H. Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 127–143. Computer Science Press, 1985.

[5] J. C. Ebergen. *Translating Programs into Delay-Insensitive Circuits.* PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1987.

[6] J. C. Ebergen. A heuristic for the design of speed-independent circuits. Personal Memorandum, 1989.

[7] M. Hennessy. *Algebraic Theory of Processes.* Series in Foundations of Computing. The MIT Press, Cambridge, Mass., 1988.

[8] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[9] He Jifeng, M. B. Josephs, and C. A. R. Hoare. A theory of synchrony and asynchrony. In *Proceedings IFIP Working Conference on Programming Concepts and Methods*, to appear, 1990.

[10] M. B. Josephs, C. A. R. Hoare, and He Jifeng. A theory of asynchronous processes. *J. ACM*, (submitted), 1989.

[11] A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[12] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. Technical Report Caltech-CS-TR-89-1, Caltech Computer Science, 1989.

[13] A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. *Theoretical Computer Science*, 60(2):177–229, 1988.

[14] I. E. Sutherland. Micropipelines. *Comm. ACM*, 32(6):720–738, 1989. Turing Award Lecture.

[15] J. T. Udding. *Classification and Composition of Delay-Insensitive Circuits.* PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1984.

[16] J. T. Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4):197–204, 1986.