An Algebra for Delay-Insensitive Circuits
(Abridged Version[*])

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

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 [13], 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:

[*]The full version of this paper appears in the ACM/AMS DIMACS series and as Technical Report WUCS-89-54, Dept. of C.S., Washington Univ., St. Louis, MO.

1. The functional behaviour of primitive delay-insensitive components can be captured by algebraic expressions.
2. All possible behaviours 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 [7]. It adapts the theory of asynchronous processes [8, 9] to the special case of delay-insensitive circuits. The possibilities of *transmission interference* and *computation interference*, characterized by Udding [14, 15], are faithfully modelled 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, 9]; the semantics is compatible with the failures-divergences model of CSP [1, 7].

Our approach complements that taken by Martin [10, 11] 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 synchronised 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 [3, 4]. 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 mechanisable, 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 ($P \sqsubseteq Q$) 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, 7], also known as the *must* ordering [6].) 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 $P \sqsubseteq Q$.

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 modelled 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. $P \sqsubseteq \bot$

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.* [14, 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 [9] 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 behaviour 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 behaviour 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 $a? \to P$. For S a finite set of alternatives, the guarded choice $[S]$ selects one of them. An alternative $a? \to P$ 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. $[a? \to P] = a?; P$

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

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

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

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. $[a? \to [S_0] \;\Box\; S_1] = [a? \to [a? \to \bot \;\Box\; S_0] \;\Box\; S_1]$

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

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

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. $[a? \rightarrow \bot \;\Box\; b? \rightarrow [a? \rightarrow P \;\Box\; S_0] \;\Box\; S_1] \;=\; [a? \rightarrow \bot \;\Box\; b? \rightarrow [S_0] \;\Box\; S_1]$

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:

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

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 *This section has been omitted.*

3 Composition

In this section we define a parallel composition operator. With it we can determine the overall behaviour of a circuit from the individual behaviour 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 [14] 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 $\mathit{skip} \rightarrow P$, we have $\mathit{skip} \rightarrow (P \parallel [S_1])$ 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 5 If one component is able to send a signal that it is unsafe for the other to receive, then their parallel composition is $\bot$.

$(a!; P) \parallel [a? \rightarrow \bot \ \Box \ S]$

$=$ { internal communication on a, Law 29. }

$P \parallel [a? \rightarrow \bot \ \Box \ S]/a?$

$=$ { Example ?? and $\bot$ null element of parallel composition, Law 28. }

$\bot$

Example 6 We compute a number of simple compositions in this example. Although the resulting behaviours 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.

$W_0 \parallel W_1$

= { definitions of W_0 and W_1 }

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

= { one choice is no choice and parallel composition through guarded choice, Law 30., using that b is internal }

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

= { internal communication on b, Law 29. }

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

= { after through prefixing }

$a?; (W_0 \parallel [b? \rightarrow \bot \;\Box\; \mathit{skip} \rightarrow c!; W_1])$

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

$a?; [\;\; a? \rightarrow ((b!; W_0) \parallel [b? \rightarrow \bot \;\Box\; \mathit{skip} \rightarrow c!; W_1])$
$\quad\;\; \Box\; \mathit{skip} \rightarrow (W_0 \parallel (c!; W_1))\;]$

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

$a?; [\mathit{skip} \rightarrow (W_0 \parallel (c!; W_1))]$

= { 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 ?? and definition of F }

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

= { 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 }

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

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

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

$=$ { 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 analyse. 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 *This section has been omitted.*

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 lead to more concise specifications and shorter proofs than would be possible using other methods. Another simplifying factor has been that, following [14], we do not distiguish 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 behaviour 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.

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. *Lect. Notes in Comp. Sci. 197*, 281–305, 1984.

[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, 174–196. Springer-Verlag, 1989.

[3] 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*, Computer Science Press, 127–143, 1985.

[4] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* PhD thesis, CMU-CS-88-119, Dept. of C.S., Carnegie-Mellon Univ., 1988.

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

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

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

[8] 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.

[9] M. B. Josephs, C. A. R. Hoare, and He Jifeng. A Theory of Asynchronous Processes. *J. ACM*, (submitted), 1989.

[10] A. J. Martin. Compiling Communicating Processes into Delay-Insensitive VLSI Circuits. *Distributed Computing*, 1:226–234, 1986.

[11] A. J. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. Caltech-CS-TR-89-1, Department of Computer Science, California Institute of Technology, 1989.

[12] A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. *Theor. Comp. Sci. 60,* 2:177-229, 1988.

[13] I. E. Sutherland. Micropipelines. 1988 Turing Award Lecture. *Communications of the ACM*, 32(6):720–738, 1989.

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

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