Specifying Confluent Processes

Olivier Tardieu and Stephen A. Edwards
Columbia University
{tardieu,sedwards}@cs.columbia.edu

Abstract

We address the problem of specifying concurrent processes that can make local nondeterministic decisions without affecting global system behavior—the sequence of events communicated along each inter-process communication channel. Such nondeterminism can be used to cope with unpredictable execution rates and communication delays.

Our model resembles Kahn's, but does not include unbounded buffered communication, so it is much simpler to reason about and implement. After formally characterizing these so-called confluent processes, we propose a collection of operators, including sequencing, parallel, and our own creation, confluent choice, that guarantee confluence by construction.

The result is a set of primitive constructs that form the formal basis of a concurrent programming language for both hardware and software systems that gives deterministic behavior regardless of the relative execution rates of the processes. Such a language greatly simplifies the verification task because any correct implementation of such a system is guaranteed to have the same behavior, a property rarely found in concurrent programming environments.

1 Introduction

We propose a formalism for specifying asynchronous, concurrent processes that, when connected, produce a global behavior that is independent of their relative execution rates. We want such a model for specifying distributed embedded systems composed of communicating hardware and software.

In an earlier work [3], we argued for a version of Kahn's influential model of dataflow processes [8]. Kahn's processes communicate exclusively through single-sender, single-receiver channels—unbounded-length FIFOs—and are specified using a sequential, imperative language that blocks when it reads from an empty FIFO. Kahn showed that networks of such processes behave deterministically, i.e., that the sequence of data values transferred over any given channel is consistent across all scheduling policies and process execution rates. In fact, Kahn established this property for a much larger class of processes—continuous functions from input histories to output histories—but provided no programming language for the specification of such processes.

We find three difficulties with Kahn's model. First, its unbounded-length buffers make simple questions undecidable, such as whether a system terminates or whether it can be executed in bounded memory. This latter problem makes practical scheduling of Kahn processes particularly difficult [14] and generally precludes a pure hardware implementation. The solution is simple: restrict communication in Kahn to be synchronous or rendezvous-style (we proposed this elsewhere [3], but were not the first). It is easy to show that this restriction does not interfere with Kahn's principle of global determinism. Furthermore, bounded buffers are easily recovered by introducing buffer processes.

The second problem is that the parallel composition of two processes cannot be represented as a sequential process. A trivial example of this is the "two wires" process in Figure 1, which simply copies its first input to its first output and its second input to its second output. Because a sequential process is forced to block on exactly one of its inputs at a time, an environment that, for instance, supplies data on the other channel only causes an deadlock. Again, the solution is straightforward: include instruction-level parallel composition in the language.

The third problem—the main one we address here—is that there are interesting processes that fit within Kahn's framework (i.e., that produce unique global behavior) yet cannot be described as a parallel composition of sequential processes, whatever the inter-process synchronizations. At the end of his 1974 paper [8], Kahn describes one such process, dubbed "warn," that emits an event as soon as there is an event available on either of the process's two inputs channels. Parallel compositions of sequential processes cannot perform such merging, yet such a process is well-behaved in Kahn's sense.

Figure 2 shows another interesting process that cannot be described as a parallel composition of sequential processes: a constructive OR gate. Its input channels a and b and output channel y convey Boolean values. A 1 value is sent on y as soon as a 1 has been seen on either a or b, otherwise 0 is sent. To make the process well-behaved, however, both inputs must be supplied before the gate can compute its next output. This eliminates the possibility of a race.

Figure 2b is an automaton for the constructive OR gate. From the initial state (in the center), the process is willing to receive on either the a or b channels. If the process receives a 0 on either channel, it goes to a state where it is waiting for

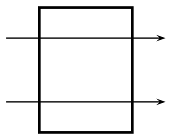
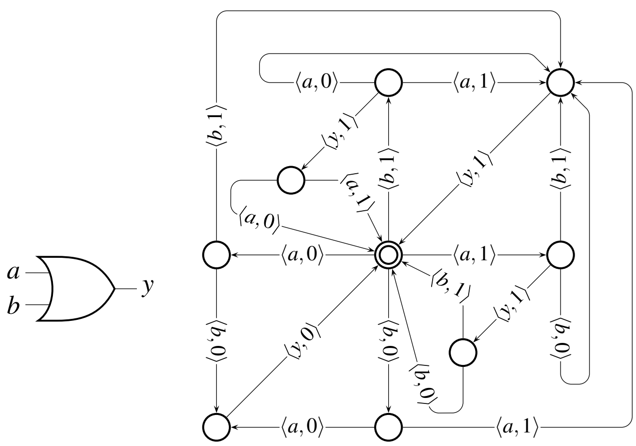


Figure 1: A process that cannot be specified using Kahn's blocking-read rule: two input channels that copy their values to two output channels. (It can be specified with a pair of concurrent processes.)



(a) (b)

Figure 2: (a) A constructive OR gate and (b) an automaton for it. The initial state is in the center. A label such as $\langle a, 0\rangle$ indicates a 0 is communicated on channel a. This drawing is arranged so that the direction an arc leaves its state indicates the type of event, e.g., moving up always means receiving 1 on b.

an input on the other channel, since the output cannot yet be established. By contrast, if the process receives a 1 on either channel, it goes to one of the two states where it can either receive a value on the other channel or emit a 1 on y. Such a choice may be dictated by the environment (e.g., it is not yet ready for an event on y), or by the process itself (e.g., the computation of the output may still be ongoing when the other input event arrives).

Kahn can represent a strict version of this process (i.e., a and b are both required before y is produced) or an asymmetric version (a is always read first, then either y is written or b is read depending on the value of a), but not the process we have described.

An implication of being able to compute such constructive functions is the ability to simulate cyclic combinational circuits [11]. Since our model is based on an assumption of delay-insensitivity, it is not surprising that we are able to correctly analyze networks of logic gates and answer whether they stabilize for all possible delay assignments (in his thesis, Shiple [15] showed that constructive simulation answers this question exactly). The behavior of our processes is, by definition, guaranteed to be the same for any delay behavior.

Others have considered simulating constructive logic in an asynchronous setting. Berry and Sentovich [2] propose a technique for simulating the Esterel language [1], which requires constructive gate evaluation. A similar technique would work in our model of computation, but ours has the advantage of guaranteeing the overall behavior is deterministic, regardless of the choice of processes; Berry and Sentovich had to choose their processes carefully.

Although it is always possible to write an automaton for a process like the constructive OR gate, not all automata are well-behaved in Kahn's sense. It is fairly easy, in fact, to perturb Figure 2 and break its determinism (e.g., change a $\langle y, 1\rangle$ to $\langle y, 0\rangle$). The regularity of Figure 2—a consequence of it being well-behaved—also suggests that higher-level constructs are appropriate.

1.1 A Roadmap

Our goal is a practical formalism for the specification of confluent processes, i.e., those that, when combined, produce the same sequence of data values through each communication channel regardless of any internal nondeterministic choices a process makes. Such choices abstractly model implementation-dependent behavior such as execution time and scheduling policy.

Our processes are objects that consume and produce atomic events provided and accepted by their environment. Inter-process communication is performed using rendezvous, i.e., both processes must agree on when data transfer occurs and no buffering occurs. This assumption of no buffers is one of the big differences between our formalism and Kahn's—by making this assumption, the scheduling problem is much easier for our processes and it opens the possibility of pure hardware implementations. Buffering can be recovered, however, by introducing buffering processes along channels.

We characterize the behavior of processes and systems using traces [21]—sets of finite sequences taken from an alphabet of events. We use traces because they are somehow operational and because they allow us to sidestep the issue of equivalent states. Events may convey data, but we always treat them as atomic.

We first define confluent processes (Section 3). The fundamental idea (Definition 3) is that a confluent process responds with the same sequence of events on each output channel when placed in a particular environment, but that the interleaving of events on different channels is irrelevant. Furthermore, the number of input events actually consumed from the environment is also consistent in this way.

This is a functional characterization in that the response of a process (Definition 1) is a function of its environment (Definition 2), which we characterize as a number and sequence of input events and opportunities for output events. That the response is a function (Theorem 1) guarantees our processes are delay-insensitive, essentially a case of the Kahn principle.

Importantly, our characterization of a process, which defines behavior for all environments, can enforce sequencing. Say we want a process to emit *a* before it emits *b*. By insisting that the process does not generate *b* when presented with no opportunity to generate *a*, we prescribe this behavior.

From the definition of confluent processes, we derive four properties about their execution (Section 4). The first property (Lemma 3) says that the order in which events are consumed and produced cannot affect future behavior (although their values can). Lemmas 4 and 5 say that once a communication event is possible, subsequent events on distinct channels cannot disable this event. Hence, it may take place now or later. For example, if a confluent process can emit a certain output at a particular point, it must eventually do so, i.e., the output cannot be suppressed by, say, an additional input. The last property (Lemma 6) tell us that output events can be postponed, but that their values may not be affected by such a delay.

Together, it turns out that these four properties exactly characterize any confluent process (Theorem 3).

In Section 5, we propose several language-level constructs, including sequencing, Kleene closure, and parallel composition, that guarantee confluence and are somehow complete, i.e., any confluent process is the result on a potentially infinite number of confluent choices between simple sequential processes. These allow us to succinctly and correctly specify confluent processes such as the constructive OR gate in Figure 2.

2 Related Work

The problem of modeling and specifying delay-independent concurrent processes has been addressed in both the software and the hardware communities. Each groups' focus is slightly different because of the physical constraints of hardware; this has lead to different techniques.

2.1 Concurrent Software

Kahn's seminal paper [8] is the cornerstone for our work, but as we explained earlier, we do not adopt it because it demands unbounded buffers and Kahn's simple sequential language cannot describe a number of useful confluent processes.

Lynch's I/O automata were also an inspiration, but Lynch also assumes unbounded buffering, although she does so by insisting that processes are always receptive to all inputs. Lynch and Stark [10] define a deterministic subset of processes that are very similar to ours and show that, when combined, they produce a deterministic system for the same reasons as Kahn, but do not suggest how to construct such automata.

The process calculi of Hoare [4] (CSP) and Milner [12] (CCS) are concerned primarily with whether two concurrent processes have the same behavior. They are concerned more with modeling a large class of systems rather than proposing restrictions that give global properties. For example, Milner explains that he began his work after discovering how difficult it was to describe the semantics of concurrent programming languages with shared variables [13, preface]. As a result, both frameworks include nondeterministic choice as a primitive operator and provide none of the determinism guarantees inherent in our model.

Milner did consider the question of determinacy and confluence [12, chap. 10], but proposed a fairly limited subset of CCS that did not address data values.

2.2 Asynchronous Hardware

The asynchronous digital hardware community has long grappled with the problem of building delay-insensitive systems. Concerned mostly with the behavior of digital logic gates, their models are necessarily lower-level than those for software.

Udding's classification of delay-insensitive behavior [17] was one of the first to provide a formal characterization of processes that are deterministic in the same sense as ours. He defines a series of properties on traces (following Van de Snepscheut's thesis work [21], which we were also inspired by) that amount to saying that changes in the order in which events arrive somehow cannot affect long-term behavior. As is appropriate for gate-level behavior, Udding only considers pure events (i.e., voltage transitions) and thus models data only as interaction order. Delay-insensitive algebra [9, 7], which discusses CSP-like processes, derived from this work.

Josephs's deterministic receptive processes [6], a subset of his receptive processes [5], share much with our model, but also differ significantly. Like Udding, Josephs only models pure events, not data. Similarly, Josephs uses a series of axioms like Udding's to characterize processes; we are able to derive roughly the same properties from a single axiom (i.e., that a process behaves functionally: the same environment will produce the same behavior), suggesting that our definition is somehow more fundamental.

Berkel [18, 19, 20] proposed using a library of delay-insensitive processes (more precisely, handshake circuits) to implement an imperative, sequential language. As in our model, the detailed behavior of the system may vary because of differing delays, but the overall system behavior is guaranteed to be consistent. This approach has also been followed by others, such as Smith and Zwarico [16].

3 Confluent Processes

Our systems resemble Kahn's: a group of concurrently-running processes that communicate data tokens through single-sender, single-receiver channels. The topology of the processes and channels are fixed before the system starts running. Unlike Kahn, however, our processes communicate in a CSP-like rendezvous style, meaning that both sender and receiver must agree on when data is to be exchanged. As mentioned earlier, this means communication in our model does not introduce unboundedness. Combining our processes, provided they are bounded, gives a bounded system.

We focus on characterizing the behavior of a single process in some environment that consists of other concurrently-running processes. Our model is abstract in that we do not model the inner workings of a process, only its interaction with its environment, i.e., the sequence of communication events it attempts to engage in.

We want to provide some flexibility in how our systems are implemented. Specifically, we do not want to have to precisely control the relative execution rates of our processes or the time taken by each communication event, but we do want to be able to describe causal relationships, e.g., that an event on channel b occurs only after the event on channel a has completed.

Our goal is to ensure the behavior of the overall system is the same for any choice of relative execution rates. Precisely, we guarantee that the number and sequence of data values communicated over each channel is the same in any valid implementation of our systems, but our assumption of uncontrolled execution rates implies that we do not consider the interleaving of events on distinct channels.

We describe process behavior using traces because they are operational and natural for describing the sequential nature of both hardware and software—our target implementation media.

Most of the machinery below attempts to characterize our notion of delay-independence on the delay-sensitive model of traces. For example, our definition of histories removes the relative order of events on different channels, similarly, our definition of environment characterizes what a process "sees" of its environment—sequences of input values and opportunities for output events.

Our definition of processes leads to a functional characterization that is similar to Kahn's [8], but differs in one important respect: Kahn assumes the environment of a process is always willing to accept additional data—a side-effect of his assumption of unbounded buffers—whereas we prescribe exactly how much data the environment of a process is willing to consume.

Sequences For a non-empty alphabet A, A^* denotes the set of all *finite-length sequences* of elements of A, including the *empty sequence*, which is denoted by ε. Concatenation of sequences is denoted by juxtaposition. We consider the usual *prefix* partial ordering $\sqsubseteq$ on sequences. For a sequence $s \in A^*$, we denote $|s| \in \mathbb{N}$ the *length* of s. For a subset $S \subseteq A^*$, we say the sequence $s \in S$ is *maximal in S* iff $\forall s' \in S : s \sqsubseteq s' \Rightarrow s = s'$.

Channels, Messages, Traces Let V be a non-empty countable set of *data values*; let I be a finite set of *input channels*; let O be a finite set of *output channels*. We require I, O, and V to be pairwise disjoint and the set of channels $C_{IO} = I \cup O$ to be non-empty.

For the OR example in Figure 2, which we will use as a running example to illustrate our definitions, $I = \{a, b\}$, $O = \{y\}$, $C_{IO} = \{a, b, y\}$, and $V = \{0, 1\}$.

An element of the set $M_{IO} = C_{IO} \times V$ denotes a *message* $\langle c, v\rangle$ of *value* v carried on *channel* c. The elements of $T_{IO} = (M_{IO})^*$ are called *traces*. In the sequel, V is constant whereas I and O may vary, hence the subscript notation.

Traces represent possible execution sequences of our processes. The trace $\langle b,1\rangle\langle a,1\rangle\langle y,1\rangle\langle a,1\rangle\langle b,0\rangle\langle y,1\rangle$ is a valid one for Figure 2 (because there is such a path) whereas $\langle y,0\rangle\langle y,1\rangle$ is not (the specification requires the first event to be on a or b).

Definition 1 (History)**.** *The* history *of a trace $t \in T_{IO}$ is the function $\bar{t} : C_{IO} \to V^*$ that maps each channel c to the sequence of values carried on c in t:*

$$\begin{aligned} \bar{\varepsilon}(c) &= \varepsilon \\ \overline{\langle c,v\rangle t}(c) &= v\bar{t}(c) \\ \overline{\langle d,v\rangle t}(c) &= \bar{t}(c) \qquad \textit{if } c \neq d \end{aligned}$$

The history of a trace preserves the number and order of messages on each channel (including values), but discards information about the interleaving of messages on different channels. This is exactly our notion of delay-insensitivity; later, we will insist that a process must respond with a unique history when placed in a particular environment.

The history of a particular trace from Figure 2 is

$$\begin{aligned} \overline{\langle b,1\rangle\langle a,1\rangle\langle y,1\rangle\langle a,1\rangle\langle b,0\rangle\langle y,1\rangle}(a) = 11 \\ \overline{\langle b,1\rangle\langle a,1\rangle\langle y,1\rangle\langle a,1\rangle\langle b,0\rangle\langle y,1\rangle}(b) = 10 \\ \overline{\langle b,1\rangle\langle a,1\rangle\langle y,1\rangle\langle a,1\rangle\langle b,0\rangle\langle y,1\rangle}(y) = 11 \end{aligned} \quad .$$

We denote H_{IO} the set of histories, i.e., the set of all functions from C_{IO} to V^*. For $h, h' \in H_{IO}$, we define $h \sqsubseteq h'$ iff $\forall c \in C_{IO} : h(c) \sqsubseteq h'(c)$. In addition, we define hh' as the function in H_{IO} such that $\forall c \in C_{IO} : hh'(c) = h(c)h'(c)$. The projection $t \mapsto \bar{t}$ is monotonic and distributes over concatenation: $\forall t, t' \in T_{IO} : \overline{tt'} = \bar{t}\,\bar{t'}$.

Intuitively, $h \sqsubseteq h'$ means that h could evolve into h'. For example, if

$$\begin{array}{lclcl} h(a) = 1 & & h'(a) = 1 & & hh'(a) = 11 \\ h(b) = 0 & \text{and} & h'(b) = 01 & \text{then } h \sqsubseteq h' \text{ and} & hh'(b) = 001 \\ h(y) = 1 & & h'(y) = 11 & & hh'(y) = 111 \end{array} \quad . \tag{1}$$

Definition 2 (Environment)**.** *An element of $E_{IO} = (I \to V^*) \times (O \to \mathbb{N})$ is an* environment*. An environment provides available input sequences and available output slots. If $e = (e_I, e_O) \in E_{IO}$, we write $e(c)$ for either $e_I(c)$ if $c \in I$ or $e_O(c)$ if $c \in O$.*

One possible environment for the process in Figure 2 is

$$\begin{array}{ll} e(a) = 010 & \text{(a sequence of 0's and 1's)} \\ e(b) = 1000 & \text{(a sequence of 0's and 1's)} \\ e(y) = 5 & \text{(an integer)} \end{array} \quad .$$

This environment is willing to supply three events on a: 0, then 1, then 0, the four-event sequence 1000 on b, and is

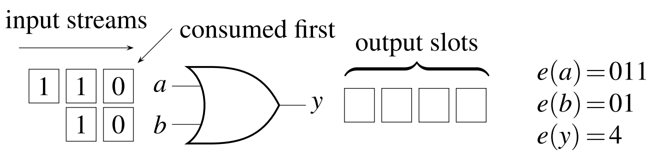


Figure 3: The constructive OR gate of Figure 2 placed in an environment that provides the sequence 011 on input a, the sequence 01 on input b, and will accept four values on y. The process will respond by consuming all the inputs and produce the sequence 011 on the output, leaving one of the output slots unused. Note that the first value in each input stream is written on the right, the opposite of the convention for sequences.

willing to accept up to 5 events on the y output channel. See Figure 3 for a graphical depiction of another environment.

For $e, e' \in E_{IO}$, we define $e \sqsubseteq e'$ iff $\forall c \in I : e(c) \sqsubseteq e'(c)$ and $\forall c \in O : e(c) \leq e'(c)$. We say that $e, e' \in E_{IO}$ are *compatible* and write $e \asymp e'$ iff $\exists s \in E_{IO} : e \sqsubseteq s \wedge e' \sqsubseteq s$. In addition, we define ee' as the function in E_{IO} such that $\forall c \in I : ee'(c) = e(c)e'(c)$ and $\forall c \in O : ee'(c) = e(c) + e'(c)$.

As for histories, $e \sqsubseteq e'$ means that inputs and output slots can be added to the environment e to give e'. When $e \asymp e'$, nothing in e is incompatible with e' and vice versa, so it is possible to find an environment that has more behavior than each. In general, this requires either $e(c) \sqsubseteq e'(c)$ or $e'(c) \sqsubseteq e(c)$ for each c.

Consider the environments

$$\begin{array}{l} e(a) = 010 \\ e(b) = 1000 \\ e(y) = 5 \end{array}, \quad \begin{array}{l} e'(a) = 01011 \\ e'(b) = 10000 \\ e'(y) = 7 \end{array} \quad \text{and} \quad \begin{array}{l} e''(a) = 0 \\ e''(b) = 10000111 \\ e''(y) = 4 \end{array}.$$

We have $e \sqsubseteq e'$ because $010 \sqsubseteq 01011$, $1000 \sqsubseteq 10000$, and $5 \leq 7$, i.e., e can be extended to become e'. Although $e' \not\sqsubseteq e''$ and $e'' \not\sqsubseteq e'$, $e' \asymp e''$ because the environment

$$\begin{array}{l} s(a) = 01011 \\ s(b) = 10000111 \\ s(y) = 7 \end{array}$$

satisfies $e' \sqsubseteq s$ and $e'' \sqsubseteq s$.

For $h \in H_{IO}$, we define $\overline{h}$ as the function in E_{IO} such that $\forall c \in I : \overline{h}(c) = h(c)$ and $\forall c \in O : \overline{h}(c) = |h(c)|$. The projection $h \mapsto \overline{h}$ abstracts output channel values, only retaining the number of values per output channel. It is monotonic and distributes over concatenation.

Projecting the histories in (1) gives the environments

$$\begin{array}{l} \overline{h}(a) = 1 \\ \overline{h}(b) = 0 \\ \overline{h}(y) = 1 \end{array}, \quad \begin{array}{l} \overline{h'}(a) = 1 \\ \overline{h'}(b) = 01 \\ \overline{h'}(y) = 2 \end{array} \quad \text{and} \quad \begin{array}{l} \overline{hh'}(a) = 11 \\ \overline{hh'}(b) = 001 \\ \overline{hh'}(y) = 3 \end{array}.$$

If $t \in T_{IO}$ then $\bar{t} \in H_{IO}$, thus $\bar{\bar{t}} \in E_{IO}$. Intuitively, $\bar{\bar{t}}$ expresses the resources consumed by t. We say that $t \in T_{IO}$ *complies with* the environment $e \in E_{IO}$ iff $\bar{\bar{t}} \sqsubseteq e$.

For the trace $t = \langle b,1\rangle\langle a,0\rangle\langle y,1\rangle\langle b,0\rangle\langle a,1\rangle\langle y,1\rangle\langle a,0\rangle$,

$$\begin{array}{l} \bar{t}(a) = 010 \\ \bar{t}(b) = 10 \\ \bar{t}(y) = 11 \end{array} \quad \text{and} \quad \begin{array}{l} \bar{\bar{t}}(a) = 010 \\ \bar{\bar{t}}(b) = 10 \\ \bar{\bar{t}}(y) = 2 \end{array}.$$

As always, both ε and t itself comply with $\bar{\bar{t}}$, but also $\langle a,0\rangle$, $\langle b,1\rangle\langle a,0\rangle\langle a,1\rangle\langle y,1\rangle$, and many others.

Confluent Languages Let $L \subseteq T_{IO}$ be a set of traces, i.e., a *language*. For $t \in L$ and $e \in E_{IO}$, we say that *t saturates e in L* and write $t \lceil L \rfloor e$ iff t complies with e and $\forall t' \in L : t \sqsubseteq t' \wedge \bar{\bar{t'}} \sqsubseteq e \Rightarrow t = t'$, that is to say iff t is maximal in $\{t' \in L : \bar{\bar{t'}} \sqsubseteq e\}$.

In general, $t \lceil L \rfloor r$ does not imply $\bar{\bar{t}} = e$ but only $\bar{\bar{t}} \sqsubseteq e$. That is, a process may engage in no more events than the environment allows, but the process may end up performing fewer.

We introduce the notion of saturation to distinguish a transient state of a process (i.e., one in which additional communication is pending) from a quiescent one. Ultimately, we want any process, when placed in a particular environment, to only have one behavior (i.e., produce a single history), but while a process is running it will pass through other, lesser histories. However, we are only concerned with the final outcome—exactly the saturating traces.

Consider a small subset of the traces generated by the OR example in Figure 2 and two environments e and e':

$$L = \left\{ \begin{array}{l} \langle b,1\rangle \\ \langle b,1\rangle\langle y,1\rangle \\ \langle b,1\rangle\langle a,0\rangle \\ \langle b,1\rangle\langle a,0\rangle\langle y,1\rangle \end{array} \right\}, \quad \begin{array}{l} e(a) = 0 \\ e(b) = 1 \\ e(y) = 0 \end{array} \quad \text{and} \quad \begin{array}{l} e'(a) = 01 \\ e'(b) = 11 \\ e'(y) = 7 \end{array}.$$

We have $\langle b,1\rangle\langle a,0\rangle \lceil L \rfloor e$ because there is no trace that extends $\langle b,1\rangle\langle a,0\rangle$ in L that still complies with e. Note that although $\langle b,1\rangle\langle a,0\rangle\langle y,1\rangle$ extends this trace, it does not comply with e, which does not allow any events on y. The trace $\langle b,1\rangle$ does not saturate e in L because it can be extended to (is a prefix of) $\langle b,1\rangle\langle a,0\rangle \in L$.

Because e' allows up to 7 y events, $\langle b,1\rangle\langle a,0\rangle$ does not saturate e' in L, but we do have $\langle b,1\rangle\langle a,0\rangle\langle y,1\rangle \lceil L \rfloor e'$. Note that no trace in this small L includes the 2 a events, 2 b events, and 7 y events allowed by the environment.

That we were able to find longer traces that did saturate these environments starting from shorter, compliant traces is no accident, as the following lemma shows:

Lemma 1. *If $L \subseteq T_{IO}$, $t \in L$, $e \in E_{IO}$, and t compiles with e then there exists $t' \in L$ such that $t \sqsubseteq t'$ and t' saturates e in L.*

Proof. By contradiction. If t does not saturate e in L, there must exist a t' complying with e in L such that $t \sqsubseteq t'$ and $t' \neq t$. By hypothesis, t' does not saturate e in L. By induction, starting from t, there exists in L a strictly increasing chain of traces compliant with e. Contradiction. □

Definition 3 (Confluent language)**.** *The language $L \subseteq T_{IO}$ is* confluent *iff it is* non-empty*,* prefix-closed*, and, for all $e \in E_{IO}$, all traces saturating e in L have the same history: $\forall e \in E_{IO}, \forall t,t' \in L : t\lceil L \rfloor e \wedge t'\lceil L \rfloor e \Rightarrow \bar{t} = \overline{t'}$.*

Intuitively, a confluent process, when placed in a particular environment, will always do the same thing, i.e., produce and consume the same number (and values) of events on each channel. The prefix-closed restriction simply guarantees that the process can proceed by single communication events. As mentioned above, we restrict our consideration to saturating traces since a process passes through many intermediate states while completing its behavior.

That the confluence restriction considers all possible environments lets us recover sequencing. For a particular environment, the confluence restriction says nothing about the order of events on different channels, suggesting that, say, a process that must emit on channel a before emitting on channel b could not be specified. However, by saying that in an environment where only a communication on b is allowed that the process will do nothing, such sequential behavior can be recovered.

We call our processes confluent for the following reason. Consider an environment e and a trace t from the language that does not saturate e. Such a trace corresponds to the behavior of a process that has not yet done all it can in e. Now, there may be two or more saturating traces, say t' and t'' that comply with e and extend t, i.e., $t'\lceil L \rfloor e$ and $t''\lceil L \rfloor e$ with $t \sqsubseteq t'$ and $t \sqsubseteq t''$. Intuitively, this means the process has a choice of what to do after t.

Our definition of confluence insists that the histories of these two traces are the same, i.e., $\overline{t'} = \overline{t''}$, i.e., that even though the process had a choice of what to do after t that it ultimately produce the same behavior, i.e., consumes the same number of inputs and produces the same number and value of outputs. This is a diamond-like confluence property: having a choice ultimately does not matter: the behavior must be the same in the end.

Consider the languages

$$L = \left\{ \begin{array}{l} \varepsilon \\ \langle b,1\rangle \\ \langle b,1\rangle\langle a,0\rangle \\ \langle b,1\rangle\langle a,0\rangle\langle y,1\rangle \\ \langle a,0\rangle \\ \langle a,0\rangle\langle b,1\rangle \\ \langle a,0\rangle\langle b,1\rangle\langle y,1\rangle \end{array} \right\} \text{ and } L' = \left\{ \begin{array}{l} \varepsilon \\ \langle b,1\rangle \\ \langle b,1\rangle\langle a,0\rangle \\ \langle b,1\rangle\langle a,0\rangle\langle y,1\rangle \\ \langle a,0\rangle \\ \langle a,0\rangle\langle b,1\rangle \\ \langle a,0\rangle\langle b,1\rangle\langle y,0\rangle \end{array} \right\}.$$

Both L and L' are non-empty and prefix-closed, but only L is confluent. To see why, consider the environment

$$\begin{array}{l} e(a) = 0 \\ e(b) = 1 \\ e(y) = 1 \end{array}.$$

In L, two traces saturate e, i.e., $\langle b,1\rangle\langle a,0\rangle\langle y,1\rangle\lceil L \rfloor e$ and $\langle a,0\rangle\langle b,1\rangle\langle y,1\rangle\lceil L \rfloor e$, and

$$\begin{array}{l} \overline{\langle b,1\rangle\langle a,0\rangle\langle y,1\rangle}(a) = \overline{\langle a,0\rangle\langle b,1\rangle\langle y,1\rangle}(a) = 0 \\ \overline{\langle b,1\rangle\langle a,0\rangle\langle y,1\rangle}(b) = \overline{\langle a,0\rangle\langle b,1\rangle\langle y,1\rangle}(b) = 1 \\ \overline{\langle b,1\rangle\langle a,0\rangle\langle y,1\rangle}(y) = \overline{\langle a,0\rangle\langle b,1\rangle\langle y,1\rangle}(y) = 1 \end{array}.$$

Again, two traces saturate e in L', i.e., $\langle b,1\rangle\langle a,0\rangle\langle y,1\rangle\lceil L' \rfloor e$ and $\langle a,0\rangle\langle b,1\rangle\langle y,0\rangle\lceil L' \rfloor e$, however, $\overline{\langle b,1\rangle\langle a,0\rangle\langle y,1\rangle}(y) = 1$ and $\overline{\langle a,0\rangle\langle b,1\rangle\langle y,0\rangle}(y) = 0$. Intuitively, L' is not confluent because it produces a different value on y depending on the order in which it sees the same events on a and b.

In a confluent language, the order of inputs on different channels may not suppress outputs, but their values may. Consider

$$L = \left\{ \begin{array}{l} \varepsilon \\ \langle b,1\rangle \\ \langle b,1\rangle\langle a,0\rangle \\ \langle b,1\rangle\langle a,0\rangle\langle y,1\rangle \\ \langle b,0\rangle \end{array} \right\} \text{ and } L' = \left\{ \begin{array}{l} \varepsilon \\ \langle b,1\rangle \\ \langle b,1\rangle\langle a,0\rangle \\ \langle b,1\rangle\langle a,0\rangle\langle y,1\rangle \\ \langle a,0\rangle \end{array} \right\}$$

in the environments

$$\begin{array}{l} e(a) = 0 \\ e(b) = 1 \\ e(y) = 1 \end{array} \text{ and } \begin{array}{l} e'(a) = 0 \\ e'(b) = 0 \\ e'(y) = 1 \end{array}.$$

It turns out L is confluent but that L' is not. In L, only one trace, $\langle b,1\rangle\langle a,0\rangle\langle y,1\rangle$ saturates e and one trace, $\langle b,0\rangle$ saturates e'. However, in L', both $\langle b,1\rangle\langle a,0\rangle\langle y,1\rangle$ and $\langle a,0\rangle$ saturate e, and they have different histories: $\overline{\langle b,1\rangle\langle a,0\rangle\langle y,1\rangle}(y) = 1$ yet $\overline{\langle a,0\rangle}(y) = \varepsilon$.

Behaviors We now formally establish that the response of a confluent process is a function of its environment.

Theorem 1. *If L is confluent then for all $e \in E_{IO}$, the set $\{h \in H_{IO} : \exists t \in T_{IO} : \bar{t} = h \wedge t\lceil L \rfloor e\}$ is a singleton, which we denote $\hat{L}(e)$. We say the function $\hat{L} : E_{IO} \to H_{IO}$ is the* behavior *of the confluent language L.*

Proof. By definition of confluent languages, this set is at most a singleton. By Lemma 1 applied to the empty trace, this set is non-empty. □

In other words, the communications of a confluent process in a deterministic environment are deterministic in the sense that the sequence of values exchanged on each channel is deterministic, regardless of local nondeterministic choices made by the process.

Moreover, behaviors characterize confluent languages.

Theorem 2. *If $L_1, L_2 \subseteq T_{IO}$ are confluent and $L_1 \neq L_2$ then $\widehat{L_1} \neq \widehat{L_2}$.*

Proof. There exists $t \in T_{IO}$ such that $t \in L_1$ and $t \notin L_2$ or vice versa. Let t' be the largest prefix of t contained in L_2. There exists $m \in M_{IO}$ such that $t'm \sqsubseteq t$. Hence, $t'm\lceil L_1 \rfloor \overline{\overline{t'm}}$ and $t'\lceil L_2 \rfloor \overline{\overline{t'm}}$. As a result, $\widehat{L_1}\left(\overline{\overline{t'm}}\right) = \overline{t'm}$ whereas $\widehat{L_2}\left(\overline{\overline{t'm}}\right) = \overline{t'}$. $\square$

Confluent processes can be equivalently specified using confluent languages or behaviors, as convenient. Not every function from E_{IO} to H_{IO}, however, encodes the behavior of a confluent process as behaviors have many distinctive properties.

Lemma 2. *If L is confluent then:*

- *$\hat{L}$ is monotonic: $\forall e, e' \in E_{IO} : e \sqsubseteq e' \Rightarrow \hat{L}(e) \sqsubseteq \hat{L}(e')$.*
- *$\overline{\hat{L}}$ is reductive: $\forall e \in E_{IO} : \overline{\hat{L}(e)} \sqsubseteq e$.*
- *$\overline{\hat{L}}$ is idempotent: $\forall e \in E_{IO} : \hat{L}\left(\overline{\hat{L}(e)}\right) = \hat{L}(e)$.*

Proof. If $e \sqsubseteq e'$ and $t\lceil L \rfloor e$ then $\overline{\overline{t}} \sqsubseteq e'$, thus, by Lemma 1, there exists $t' \in T_{IO}$ such that $t \sqsubseteq t'$ and $t'\lceil L \rfloor e'$. In particular, $\overline{t} \sqsubseteq \overline{t'}$, that is to say $\hat{L}(e) \sqsubseteq \hat{L}(e')$.

If $t\lceil L \rfloor e$ then $\overline{\overline{t}} \sqsubseteq e$ and $\hat{L}(e) = \overline{t}$, so $\overline{\hat{L}(e)} \sqsubseteq e$.

$\forall t \in L : t\lceil L \rfloor \overline{\overline{t}}$. In particular, if $t\lceil L \rfloor e$ then $t\lceil L \rfloor \overline{\overline{t}}$. Therefore, $\hat{L}\left(\overline{\hat{L}(e)}\right) = \hat{L}\left(\overline{\overline{t}}\right) = \overline{t} = \hat{L}(e)$. $\square$

While behaviors resemble Kahn's continuous functions from input histories to output histories, they differ significantly. Our model relies on synchronous communications. Thus, first, behaviors take into account the environment willingness to receive output events; second, behaviors report the actual sequences of input events they consume. Also, while behaviors are concerned with finite sequences only, Kahn considers both finite and infinite sequences. This, however, is a technical issue of no practical consequence, which we established in a previous work [3].

4 Properties of Confluent Languages

The following four properties hold for any confluent language $L \subseteq T_{IO}$. We introduce them to simplify later proofs and to provide more intuition about how confluent languages behave.

It turns out that these four properties *exactly* characterize confluent languages, i.e., any language that has these properties (in addition to being non-empty and prefix-closed) is confluent. We prove this below (Theorem 3).

If a confluent process can consume and produce some sequence of events on its channels, then the order in which it did so is irrelevant in the future. The following lemma formalizes this:

Lemma 3. *If $tt' \in L$, $t'' \in L$, and $\overline{\overline{t}} = \overline{\overline{t''}}$ then $t''t' \in L$.*

Proof. By induction on the length of t'. Obvious if $t' = \varepsilon$. Otherwise, there exist $m \in M_{IO}$ and $t''' \in T_{IO}$ such that $t' = t'''m$. By induction hypothesis, $t''t''' \in L$. By Lemma 1, there exists $t'''' \in T_{IO}$ such that $t''t'''t''''\lceil L \rfloor \overline{\overline{t''t'''m}}$. Since $tt'''m\lceil L \rfloor \overline{\overline{t''t'''m}}$, by confluence, $\overline{t''t'''t''''} = \overline{tt'''m}$. Hence, $t'''' = m$ and $t''t' \in L$. $\square$

The following lemma states that if at some point a process can either emit or receive an event $\langle c, v \rangle$, then if the process can do anything else at that same point, it must be able to emit or receive $\langle c, v \rangle$ in the future. Colloquially, communication is "sticky."

Lemma 4. *If $t\langle c, v\rangle \in L$, $tt' \in L$, and $\overline{t'}(c) = \varepsilon$ then $tt'\langle c, v\rangle \in L$.*

Proof. $\overline{\overline{tt'}} \sqsubseteq \overline{\overline{tt'\langle c, v\rangle}}$ and $\overline{\overline{t\langle c, v\rangle}} \sqsubseteq \overline{\overline{t\langle c, v\rangle t'}} = \overline{\overline{tt'\langle c, v\rangle}}$ since $\overline{t'}(c) = \varepsilon$. By Lemma 1, there exist t'' and $t''' \in T_{IO}$ such that $tt't''\lceil L \rfloor \overline{\overline{tt'\langle c, v\rangle}}$ and $t\langle c, v\rangle t'''\lceil L \rfloor \overline{\overline{tt'\langle c, v\rangle}}$. By confluence, $\overline{tt't''} = \overline{t\langle c, v\rangle t'''}$, hence $t'' = \langle c, v\rangle$. As a result $tt'\langle c, v\rangle \in L$. $\square$

The following lemma refines this notion further: if a communication event can be postponed as Lemma 4 allows, then it does not affect the future.

Lemma 5. *If $t\langle c, v\rangle \in L$, $tt'\langle c, v\rangle t'' \in L$, and $\overline{t'}(c) = \varepsilon$ then $t\langle c, v\rangle t't'' \in L$.*

Proof. By induction on the length of t'. Obvious it $t' = \varepsilon$. Otherwise, there exist $d \in C_{IO}$, $w \in V$, and $t''' \in T_{IO}$ such that $t' = \langle d, w\rangle t'''$, $d \neq c$, and $\overline{t'''}(c) = \varepsilon$. By Lemma 4, $t\langle d, w\rangle\langle c, v\rangle \in L$ and $t\langle c, v\rangle\langle d, w\rangle \in L$. By induction hypothesis, $t\langle d, w\rangle\langle c, v\rangle t'''t'' \in L$. By Lemma 3, $t\langle c, v\rangle\langle d, w\rangle t'''t'' \in L$ that is to say $t\langle c, v\rangle t't'' \in L$. $\square$

This next lemma says that the sequence of data values written on a particular channel is not affected by delays, an important component of our delay-insensitive philosophy.

Lemma 6. *If $t\langle c, v\rangle \in L$, $tt'\langle c, w\rangle \in L$, $\overline{t'}(c) = \varepsilon$, and $c \in O$ then $v = w$.*

Proof. $\overline{\overline{t\langle c, v\rangle}} \sqsubseteq \overline{\overline{tt'\langle c, w\rangle}}$ since $c \in O$. By Lemma 1, there exists $t'' \in T_{IO}$ such that $t\langle c, v\rangle t''\lceil L \rfloor \overline{\overline{tt'\langle c, w\rangle}}$. By confluence, $\overline{t\langle c, v\rangle t''} = \overline{tt'\langle c, w\rangle}$, which implies $v = w$. $\square$

The following theorem shows that together, the properties in Lemmas 3–6 exactly characterize confluent languages. Among other things, this means we could instead have taken the properties in Lemmas 3–6 as the definition of a confluent language and from there derived our notions of compliance, histories, and so forth. Josephs [6] takes this approach. We chose to start with our functional characterization of confluence because it more closely resembles Kahn's approach and is easier to state.

Theorem 3. *The language $L \subseteq T_{IO}$ is confluent if it is non-empty, prefix-closed, and obeys Lemmas 3 to 6.*

Proof. Let us choose $e \in E_{IO}$, t_1, t_2 saturating e in L and prove $\overline{t_1} = \overline{t_2}$. We can assume $|t_1| \leq |t_2|$.

For $0 \leq n \leq |t_1|$, we establish by induction on n the property $P(n) = \exists t_0, t_3 \in L : |t_0| = n \wedge t_0 \sqsubseteq t_1 \wedge t_0 \sqsubseteq t_3 \wedge \overline{t_2} = \overline{t_3}$.

$P(0)$ is obtained by choosing $t_0 = \varepsilon$ and $t_3 = t_2$.

For n such that $0 \leq n < |t_1|$, let us assume $P(n)$, i.e., let $t_0, t_3 \in L$ be such that $|t_0| = n$, $t_0 \sqsubseteq t_1$, $t_0 \sqsubseteq t_3$, and $\overline{t_2} = \overline{t_3}$. There exist $c,d \in C_{IO}$, $v,w \in V$, $t_1', t_3' \in T_{IO}$ such that $t_1 = t_0\langle c,v\rangle t_1'$ and $t_3 = t_0\langle d,w\rangle t_3'$. If $\langle c,v\rangle = \langle d,w\rangle$ then $P(n+1)$ holds. We now assume $\langle c,v\rangle \neq \langle d,w\rangle$ and define $t_3'' = \langle d,w\rangle t_3'$.

Since $\overline{\overline{t_3}} = \overline{\overline{t_2}}$ and $t_2\lceil L\rfloor e$, we have $t_3\lceil L\rfloor e$ by Lemma 3. By Lemma 4, if $\overline{t_3''}(c) = \varepsilon$ then $t_3\langle c,v\rangle \in L$, which contradicts $t_3\lceil L\rfloor e$. Hence, $\overline{t_3''}(c) \neq \varepsilon$. In other words, there exist $u \in V$ and $x,y \in T_{IO}$ such that $t_3'' = x\langle c,u\rangle y$ with $\overline{x}(c) = \varepsilon$. By Lemma 6, if $c \in O$ then $u = v$. Otherwise, if $c \in I$ then $\overline{\overline{t_0\langle c,v\rangle}} \sqsubseteq e$ and $\overline{\overline{t_0 x\langle c,u\rangle}} \sqsubseteq e$ with $\overline{x}(c) = \varepsilon$, so $u = v$. Hence, $u = v$ in all cases.

By Lemma 5, $t_0\langle c,v\rangle xy \in L$. Moreover, $\overline{t_0\langle c,v\rangle xy} = \overline{t_3} = \overline{t_2}$, which concludes the proof of $P(n+1)$.

For $n = |t_1|$, $P(n)$ reduces to $\exists t_3 \in L : t_1 \sqsubseteq t_3 \wedge \overline{t_2} = \overline{t_3}$. In particular, $\overline{\overline{t_3}} \sqsubseteq e$. Since $t_1\lceil L\rfloor r$, this implies $t_1 = t_3$. To conclude, $\overline{t_1} = \overline{t_3} = \overline{t_2}$. $\square$

5 Language Constructs

To this point, we have only characterized confluent languages, but have not provided a practical way to construct them. In this section, we provide a series of operators for building languages that guarantee confluent behavior.

We construct confluent languages starting with pure events and combining them sequentially (;), through repetition (Kleene-*), in parallel (||), and through "confluent choice" (|). Each of these constructs is confluence-closed (i.e., combining two confluent languages with any of these operators gives a confluent language). Furthermore, they are complete in the sense that they can be used to construct any confluent language.

We propose these operators as the building blocks of a more user-friendly language for specifying confluent processes. A practical language would also include constructs such as variables, conditional statements, scoping, and so forth; we will address this in future work.

Empty Trace We start with the most basic language. Whatever I and O, we define the language $\mathscr{E}_{IO} = \{\varepsilon\} \subseteq T_{IO}$.

Message For $\langle c,v\rangle \in M_{IO}$, we denote $\langle\langle c,v\rangle\rangle_{IO}$ the confluent language $\{\varepsilon, \langle c,v\rangle\} \subseteq T_{IO}$.

This language expresses the willingness to engage in a single communication. Because confluent languages must be prefix-closed, this language includes the empty trace ε.

The next construct is the familiar sequencing operator on traces. The only technical point here is that in in the construct $L_1;L_2$, we want L_2 to start only after L_1 has "terminated" [21]. We say L_1 has terminated when it has reached a maximal trace, i.e., a point at which the process cannot engage in any more communication. As a result, the sequential composition differs from the concatenation of languages, which typically does not preserve confluence.

Sequence The *sequential composition* $L_1;L_2$ of the languages $L_1, L_2 \subseteq T_{IO}$ is defined as follows:

$$t \in L_1;L_2 \Leftrightarrow \begin{Bmatrix} \exists t_1 \in L_1 \\ \exists t_2 \in L_2 \end{Bmatrix} : \begin{Bmatrix} t = t_1 t_2 \\ (t_1 \text{ maximal in } L_1) \vee (t_2 = \varepsilon) \end{Bmatrix}.$$

Lemma 7. *The ";" operator is associative:* $\forall L_1, L_2, L_3 \subseteq T_{IO} : (L_1;L_2);L_3 = L_1;(L_2;L_3)$.

Proof. See Van de Snepscheut [21]. $\square$

Theorem 4. *If L_1 and L_2 are confluent then $L_1;L_2$ is confluent.*

Proof. $L_1;L_2$ is non-empty and prefix-closed. Let us suppose t_1t_2 and $t_1't_2'$ saturate $e \in E_{IO}$ in $L_1;L_2$ with $t_1, t_1' \in L_1$ and $t_2, t_2' \in L_2$.

If $t_2 \neq \varepsilon$ then t_1 is maximal in L_1, thus $t_1\lceil L_1\rfloor e$. If $t_2 = \varepsilon$ then t_1 is maximal in $\{t \in L_1;L_2 : \overline{\overline{t}} \sqsubseteq e\}$, thus $t_1\lceil L_1\rfloor e$. In any case, $t_1\lceil L_1\rfloor e$ and similarly $t_1'\lceil L_1\rfloor e$ so that $\overline{t_1} = \overline{t_1'}$ by confluence of L_1.

If t_1 is maximal in L_1 but t_1' is not then there exists $t \neq \varepsilon$ such that $t_1't \in L_1$. As a result, $t_1\lceil L_1\rfloor \overline{\overline{t_1't}}$ and $t_1't\lceil L_1\rfloor \overline{\overline{t_1't}}$. Since L_1 is confluent, $\overline{t_1} = \overline{t_1't}$. Contradiction. Therefore, either both t_1 and t_1' are maximal in L_1 or neither is.

In the first case, both t_2 and t_2' saturate e' in L_2 where $e' \in E_{IO}$ is such that $e = \overline{\overline{t_1}}e'$. Since L_2 is confluent, $\overline{t_2} = \overline{t_2'}$. In the second case, $t_2 = \varepsilon = t_2'$, which implies $\overline{t_2} = \overline{t_2'}$ as well.

To conclude, $\overline{t_1t_2} = \overline{t_1't_2'}$. Hence, $L_1;L_2$ is confluent. $\square$

The next construct is the obvious infinite extension of sequencing: Kleene's * operator. It simply restarts L whenever L terminates.

Kleene Closure For $L \subseteq T_{IO}$, we define $(L^n)_{n\geq 1}$ by induction on n: $L^1 = L$, $\forall n \geq 1 : L^{n+1} = L^n;L$. We define $L^* = \bigcup_{n\geq 1} L^n$.

Theorem 5. *If L is confluent then L^* is confluent.*

Proof. $L^1 \subseteq L^2 \subseteq \cdots \subseteq L^n \subseteq \cdots \subseteq L^*$. If $t\lceil L^*\rfloor e$ and $t'\lceil L^*\rfloor e$ then there exists $n \geq 1$ such that $t \in L^n$ and $t' \in L^n$. Therefore, $t\lceil L^n\rfloor e$ and $t'\lceil L^n\rfloor e$. By Theorem 4, L^n is confluent, thus $\overline{t} = \overline{t'}$. $\square$

The next construct—the restriction operator—hides events on output channels, which is useful to hide interprocess communications. Indeed, we shall see later that inter-process communication channels in a network of processes (assembled by means of parallel compositions and confluent choices) can be observed by the environment without harm, i.e., are by default treated as output channels of the network. On the other hand, a similar operator that hid inputs would not be confluent.

We start with the restriction of traces w.r.t. both inputs and outputs as this will be useful for defining the parallel operator, then define the restriction of languages w.r.t. outputs only.

Restriction For $I' \subseteq I \cup O$ and $O' \subseteq O$ such that $I' \cup O' \neq \emptyset$ and $I' \cap O' = \emptyset$, we define the *restriction* of the trace $t \in T_{IO}$ to channels $C_{I'O'}$ as the trace $t|_{I'O'} \subseteq T_{I'O'}$:

$$\begin{array}{rcll} \varepsilon|_{I'O'} & = & \varepsilon & \\ (\langle c,v\rangle t)|_{I'O'} & = & \langle c,v\rangle (t|_{I'O'}) & \text{if } c \in C_{I'O'} \\ (\langle c,v\rangle t)|_{I'O'} & = & t|_{I'O'} & \text{if } c \notin C_{I'O'} \end{array}$$

Note that an output channel may be turned into an input channel in the process (as required for the parallel composition).

Restrictions of histories and environments can be similarly defined. Restriction commutes with projections (from traces to histories and from histories to environments) and distributes over concatenation.

For $O' \subseteq O$ and $I \cup O' \neq \emptyset$, the *restriction* $L|_{O'} \subseteq T_{IO'}$ of the language $L \subseteq T_{IO}$ to the set of output channels O' is defined by $t' \in L|_{O'} \Leftrightarrow \exists t \in L : t' = t|_{IO'}$.

Theorem 6. *If $L \subseteq T_{IO}$ is confluent, $O' \subseteq O$, and $I \cup O' \neq \emptyset$ then $L|_{O'}$ is confluent.*

Proof. If $t_1' \lceil L|_{O'} \rfloor e'$ and $t_2' \lceil L|_{O'} \rfloor e'$ for some $e' \in E_{IO'}$ then there exist $t_1, t_2 \in L$ such that $t_1' = t_1|_{IO'}$ and $t_2' = t_2|_{IO'}$. Moreover, $\overline{\overline{t_1}} \asymp \overline{\overline{t_2}}$ since t_1 and t_2 may only differ on output messages. As a result, there exists $e \in E_{IO}$ such that $\overline{\overline{t_1}} \sqsubseteq e$, $\overline{\overline{t_2}} \sqsubseteq e$, and $e' = e|_{IO'}$. Hence, there exist $t_1'', t_2'' \in L$ such that $t_1 t_1'' \lceil L \rfloor e$ and $t_2 t_2'' \lceil L \rfloor e$. By confluence of L, we obtain $\overline{t_1 t_1''} = \overline{t_2 t_2''}$, which implies $\overline{t_1'(t_1''|_{IO'})} = \overline{t_2'(t_2''|_{IO'})}$. Since $t_1 t_1''|_{IO'} \sqsubseteq e'$ and $t_2 t_2''|_{IO'} \sqsubseteq e'$, we have also $t_1''|_{IO'} = \varepsilon = t_2''|_{IO'}$. Therefore, $\overline{t_1'} = \overline{t_2'}$. $\square$

We now come to the two key constructs for composing confluent processes. The first construct—parallel composition—interleaves the execution of two processes and requires them to agree on events on shared input and output channels—a sort of "logical AND" concurrency. Later, we will introduce "logical OR" concurrency in the form of a confluent choice operator.

Figure 4 illustrates how two confluent processes behave when combined using both the parallel and confluent choice operator. Both operators make the two processes run in parallel; the difference comes in how shared input channels are split and how shared outputs are merged.

Parallel Composition The *parallel composition* $L_1||L_2$ of the languages $L_1 \subseteq T_{I_1O_1}$ and $L_2 \subseteq T_{I_2O_2}$ is the language with input channels $I = I_1 \cup I_2 \setminus (O_1 \cup O_2)$ and output channels $O = O_1 \cup O_2$ such that $t \in L_1||L_2$ iff $t|_{I_1O_1} \in L_1$ and $t|_{I_2O_2} \in L_2$.

Parallel composition combines processes as shown in Figure 4 and insists that both processes participate in any shared events. When an input channel is split, parallel composition insists that both processes receive the event for it to happen. Similarly, when an output channel is merged, an output event occurs only if both processes participate and agree on

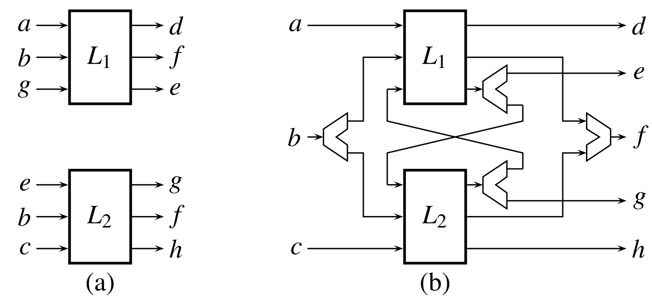


Figure 4: (a) Two confluent processes with their inputs and outputs. (b) The result of combining them in parallel or with confluent choice; the behavior of the split and merge operators (the triangular objects) distinguishes the operators. Input b is shared, as is output e. Inputs e and g are connected to the identically-named outputs on the other process, but are no longer inputs of the whole system, only outputs.

the value. If the two processes agree that an event should occur on a particular channel but disagree on the value, no event is generated and the merge effectively prevents either process from ever producing an event on that channel again.

Lemma 8. *The "$||$" operator is associative.*

Proof. See Van de Snepscheut [21]. $\square$

In general, we define the parallel composition of the potentially infinite set $\{L_n\}_{n \in S}$ of languages of respective channels $C_{I_nO_n}$ as the language with output channels $O = \bigcup_{n \in S}\{O_n\}$ and input channels $I = \bigcup_{n \in S}\{I_n\} \setminus O$ that exactly contains the traces $t \in T_{IO}$ such that $\forall n \in S : t|_{I_nO_n} \in L_n$. This extends the earlier definition for finite parallel compositions.

Theorem 7. *If $\forall n \in S : L_n$ is confluent then their parallel composition L is confluent.*

Proof. $\forall n \in S : L_n$ contains the empty trace and is prefix-closed. As a result, the parallel composition is non-empty and prefix-closed. Let us show it obeys Lemmas 3–6.

If $tt' \in L$, $t'' \in L$, and $\overline{\overline{t}} = \overline{\overline{t''}}$ then $\forall n \in S$: $t|_{I_nO_n} t'|_{I_nO_n} = tt'|_{I_nO_n} \in L_n$, $t''|_{I_nO_n} \in L_n$, and $\overline{\overline{t|_{I_nO_n}}} = \overline{\overline{t''|_{I_nO_n}}}$. Since L_n is confluent, by Lemma 3, $t''|_{I_nO_n} t'|_{I_nO_n} \in L_n$ for all n. Therefore, $t''t' \in L$.

If $t\langle c,v\rangle \in L$, $tt' \in L$, $\overline{t'}(c) = \varepsilon$, and $c \in C|_{I_nO_n}$ then $t|_{I_nO_n}\langle c,v\rangle \in L_n$, $t|_{I_nO_n} t'|_{I_nO_n} \in L_n$, and $\overline{t'|_{I_nO_n}}(c) = \varepsilon$. Since L_n is confluent, by Lemma 4, $t|_{I_nO_n} t'|_{I_nO_n}\langle c,v\rangle \in L_n$. Otherwise, if $c \notin C|_{I_nO_n}$ then $(tt'\langle c,v\rangle)|_{I_nO_n} = tt'|_{I_nO_n} \in L_n$. Therefore, $tt'\langle c,v\rangle \in L$.

If $t\langle c,v\rangle \in L$, $tt'\langle c,v\rangle t'' \in L$, $\overline{t'}(c) = \varepsilon$, and $c \in C|_{I_nO_n}$ then $t|_{I_nO_n}\langle c,v\rangle \in L_n$, $t|_{I_nO_n} t'|_{I_nO_n}\langle c,v\rangle t''|_{I_nO_n} \in L_n$, and $\overline{t'|_{I_nO_n}}(c) = \varepsilon$. Since L_n is confluent, by Lemma 5, we obtain $t|_{I_nO_n}\langle c,v\rangle t'|_{I_nO_n} t''|_{I_nO_n} \in L_n$. Otherwise, if $c \notin C|_{I_nO_n}$ then $(t\langle c,v\rangle t't'')|_{I_nO_n} = (tt'\langle c,v\rangle t'')|_{I_nO_n} \in L_n$. Therefore, $t\langle c,v\rangle t't'' \in L$.

If $t\langle c,v\rangle \in L$, $tt'\langle c,w\rangle \in L$, $\overline{t'}(c) = \varepsilon$, and $c \in O$ then let us choose $n \in S$ such that $c \in O_n$. Since $t|_{I_nO_n}\langle c,v\rangle \in L_n$, $t|_{I_nO_n}t'|_{I_nO_n}\langle c,w\rangle \in L_n$, and $\overline{t'|_{I_nO_n}}(c) = \varepsilon$, we conclude $v = w$ by applying Lemma 6 to L_n.

Thanks to Theorem 3, the parallel composition is confluent. □

Importantly, if all L_n share the same input and output channels then their parallel composition is their intersection.

Our final challenge is to define confluent choice. The usual choice operator in regular expressions does not usually produce confluent languages because it suppresses the untaken alternative.

Instead, we define the confluent choice of two languages as the least confluent language that contains the behavior of both. This can be thought of as the usual choice operator followed by a closure operation that adds the behavior required by confluence.

First, we prove a theorem that ensures us that if we take a subset of a confluent language, there is always a unique, least way to "grow" it back into a confluent language. We need this for the definition of confluent choice because we need to ensure that there is such a confluent language.

Theorem 8. *If $L \subseteq L' \subseteq T_{IO}$ and L' is confluent then there exists a least confluent language $L_0 \subseteq T_{IO}$ that contains L: $\forall L'_0 \subseteq T_{IO} : L \subseteq L'_0 \wedge L'_0$ confluent $\Rightarrow L_0 \subseteq L'_0$.*

Proof. The set of all confluent languages containing L is non-empty thanks to L'. Let L_0 be the intersection (i.e., the parallel composition) of all such languages. By Theorem 7, L_0 is confluent. By construction, it is contained in any confluent language containing L. □

Another challenging aspect of confluent choice is that it insists on a form of compatibility between the processes being combined. Indeed, not every language is contained in a confluent language. For instance, for values $V = \{0,1\}$ and channels $I = \emptyset$ and $O = \{c\}$, there exists no confluent language containing the two traces $\langle c,0\rangle$ and $\langle c,1\rangle$ since any such language would violate Lemma 6.

Intuitively, two languages are compatible if they do not contradict each other when generating shared output events. Technically, we insist that the combination of the traces in the languages we want to combine via confluent choice are consistent in the following sense.

Consistency We say that a language is *consistent* iff it is contained in a confluent language.

Intuitively, a consistent language may be missing behavior that would be required to make it confluent (e.g., the process stops even though it is obligated to generate addition outputs in compliance with Lemma 4) but it does nothing that prevents confluence (e.g., produces a conflicting output in response to a different order in which inputs arrived).

The language consisting of the empty trace is the shortest consistent language.

Confluent Choice For two languages $L_1, L_2 \subseteq T_{IO}$, if $L_1 \cup L_2$ is consistent, we define the *confluent choice* $L_1|L_2$ as the least confluent language containing L_1 and L_2.

Lemma 9. *The "$|$" operator is associative. $\forall L_1, L_2, L_3 \subseteq T_{IO} : (L_1|L_2)|L_3$ and $L_1|(L_2|L_3)$ are either both defined or both undefined, and equal if defined.*

Proof. If $L_1 \cup L_2 \cup L_3$ is contained in the confluent language L then both $L_1|L_2$ and $L_2|L_3$ are defined and contained in L. Therefore, both $(L_1|L_2)|L_3$ and $L_1|(L_2|L_3)$ are defined. Moreover, $L_1 \subseteq L_1|L_2 \subseteq (L_1|L_2)|L_3$. $L_2 \subseteq L_1|L_2 \subseteq (L_1|L_2)|L_3$. $L_3 \subseteq (L_1|L_2)|L_3$. Hence, $L_1|(L_2|L_3) \subseteq (L_1|L_2)|L_3$ and vice versa.

Otherwise, if $L_1 \cup L_2 \cup L_3$ is inconsistent, then either $L_1|L_2$ is undefined or $(L_1|L_2) \cup L_3$ is inconsistent. Similarly, either $L_2|L_3$ is undefined or $L_1 \cup (L_2|L_3)$ is inconsistent. □

In general, we consider the union $\bigcup_{n\in S}\{L_n\}$ of a potentially infinite set of languages of respective channels $C_{I_nO_n}$ to be the language with output channels $O = \bigcup_{n\in S}\{O_n\}$ and input channels $I = \bigcup_{n\in S}\{I_n\} \setminus O$ that contains the traces $t \in T_{IO}$ such that $\exists n \in S : t \in L_n$.

If this union is consistent, we define the confluent choice of the set of languages as the least confluent language containing the union. This extends the earlier definition for finite confluent choices.

5.1 Comparison

Once again, the basic idea behind confluent choice is to merge the languages, provided they are not inconsistent, then add whatever behavior is necessary to make the result confluent. Lemmas 3–6 suggest the sort of behavior that must be added. For example, if an input or output is allowed at a certain point then it must also be allowed later.

When two processes do not share any inputs or outputs, confluent choice is the same as parallel, i.e., the same as interleaving; it is only when processes share input or output channels that the difference arises.

Intuitively, the parallel operator imposes "logical AND" concurrency, meaning that both processes must participate in all events on shared channels. By contrast, the confluent choice operator is something like "logical OR" concurrency: it allows either process to consume or produce an event on a shared channel, but to maintain confluence, the event is still available to the other process, which may ignore this event forever (and stop using the corresponding channel) or consume it eventually.

Like parallel composition, confluent choice combines processes as shown in Figure 4, but the split and merge behavior is very different than the parallel case.

Splitting an input in confluent choice effectively buffers it. The same sequence is fed to both processes, but the two are not required to remain synchronized. Specifically, one process is allowed to get arbitrarily far ahead of the other in reading the sequence; both processes see the same sequence of events on the channel.

Confluent choice, then, is a potentially dangerous operator in that it can require unbounded resources. However, this is not always the case. For example, if the length of the sequences accepted by the two processes is bounded, it follows that the size of the splitter buffer is bounded. While this seem overly restrictive, observe that enclosing such a pair of processes in a Kleene-* construct does not require an infinite buffer. In general, unbounded buffers are only required when one process can get arbitrarily far ahead of the other, which we do not expect to be very common in practice.

Merging outputs in confluent choice is similar. The processes can get arbitrarily out-of-sync, but must agree on the values being sent on the channel. This is guaranteed if the union of the two languages is consistent, so the behavior of the merge is simply to keep track of how many events the two processes have generated on the merged output channel and transmit the longer sequence. Again, this may require unbounded resources (i.e., to keep track of a potentially unbounded difference in the number of events), but if the two processes have bounded-length traces, it is possible to bound the maximum number that the merge operation must track. Note that unlike the split, which must store the data values, the merge only needs to maintain a count.

Examples To illustrate confluent choice and parallel composition, first consider the two languages $A|B$ and $A||B$ with $A = \langle\langle a,0\rangle\rangle_{\{a\}\emptyset}$ and $B = \langle\langle b,0\rangle\rangle_{\{b\}\emptyset}$, that it to say A has one input channel a and no output channel and similarly for B.

To simplify the notation, we will drop the angle bracket notation and denote $\langle a,0\rangle$ by a and $\langle b,0\rangle$ by b. Hence, $A = \{\varepsilon, a\}$ and $B = \{\varepsilon, b\}$.

The confluent choice and parallel composition of A and B are equal:

$$A|B = A||B = \{\varepsilon, a, b, ab, ba\}$$

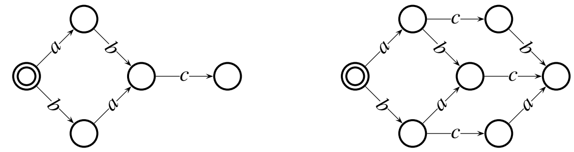
The parallel operator requires that any trace in $A||B$, when restricted to the channels in A, must be a trace in A, and similarly for B.

For $A|B$, first note that $A \cup B = \{\varepsilon, a, b\}$. This is not confluent, but it is consistent because it is contained in $A||B$, which is confluent. Moreover, neither $\{\varepsilon, a, b, ab\}$ nor $\{\varepsilon, a, b, ba\}$ is confluent. Therefore, $A|B = A||B$.

Now, to see the difference between confluent choice and parallel composition, consider the languages $X|Y$ and $X||Y$ with $X = \langle\langle a,0\rangle\rangle_{\{a,c\}\emptyset};\langle\langle c,0\rangle\rangle_{\{a,c\}\emptyset}$ and $Y = \langle\langle a,0\rangle\rangle_{\{b,c\}\emptyset};\langle\langle c,0\rangle\rangle_{\{b,c\}\emptyset}$, that is to say $X = \{\varepsilon, a, ac\}$ with input channels $\{a,c\}$ and no output channels and $Y = \{\varepsilon, b, bc\}$ with input channels $\{b,c\}$ and no output channels. Here, the result is quite different as illustrated in Figure 5:

$$X|Y = \{\varepsilon, a, b, ac, abc, acb, bc, bca, bac\}$$
$$X||Y = \{\varepsilon, a, b, ab, ba, abc, bac\}$$

Intuitively, the confluent choice operator allows the c event to be generated independently, i.e., before the other process has also decided to accept or generate it. Parallel



(a) (b)

Figure 5: Two automata generating the traces of the (a) parallel composition and (b) confluent choice of the languages $\{\varepsilon, a, ac\}$ with channels $\{a,c\}$ and $\{\varepsilon, b, bc\}$ with channels $\{b,c\}$.

composition, by contrast, always insists that the two processes agree on events before they are visible to the environment.

Last example, consider the languages A^p and A^q for $p,q \in \mathbb{N}$, that is to say A^p is the set of traces containing up to p events a, and similarly for A^q. Then,

$$A^p;A^q = A^{p+q} \qquad A^p \cup A^q = A^{\max(p,q)}$$
$$A^p||A^q = A^{\min(p,q)} \qquad A^p|A^q = A^{\max(p,q)}$$

Constructive OR gate With the confluent choice operator, we are able to succinctly express the behavior of the constructive OR gate in Figure 2, for instance as:

$$(\langle\langle a,1\rangle\rangle;\langle\langle y,1\rangle\rangle|\langle\langle b,1\rangle\rangle;\langle\langle y,1\rangle\rangle|\langle\langle a,0\rangle\rangle;\langle\langle b,0\rangle\rangle;\langle\langle y,0\rangle\rangle|\langle\langle b,0\rangle\rangle)^*$$

deciding the ";" operator binds tighter than "|", and considering that all the combined languages have input channels $\{a,b\}$ and output channels $\{y\}$.

This specification consists of the Kleene closure (the *) of the confluent choice among four little languages. The first two languages, $\langle\langle a,1\rangle\rangle;\langle\langle y,1\rangle\rangle$ and $\langle\langle b,1\rangle\rangle;\langle\langle y,1\rangle\rangle$, are symmetric and handle the early generation of y. Each waits for a 1 on a or b, then generates a 1 on y in response. Because they are combined using confluent choice, only a single y event is ever generated, i.e., if the environment provides both $\langle a,1\rangle$ and $\langle b,1\rangle$, both processes generate $\langle y,1\rangle$, and confluent choice merges them.

The confluent choice between the third and fourth languages, $\langle\langle a,0\rangle\rangle;\langle\langle b,0\rangle\rangle;\langle\langle y,0\rangle\rangle$ and $\langle\langle b,0\rangle\rangle$, waits for zeros on both a and b, then produces a 0 on y. Indeed, while the third language requires $\langle a,0\rangle$ to be read first, the fourth language allows $\langle b,0\rangle$ to come first. In fact, this confluent choice is equivalent to each of the following languages:

- $(\langle\langle a,0\rangle\rangle;\langle\langle b,0\rangle\rangle;\langle\langle y,0\rangle\rangle)|(\langle\langle b,0\rangle\rangle;\langle\langle a,0\rangle\rangle;\langle\langle y,0\rangle\rangle)$
- $(\langle\langle a,0\rangle\rangle_{\{a\}\{y\}}||\langle\langle b,0\rangle\rangle_{\{b\}\{y\}});\langle\langle y,0\rangle\rangle_{\{a,b\}\{y\}}$
- $(\langle\langle a,0\rangle\rangle_{\{a\}\{y\}};\langle\langle y,0\rangle\rangle_{\{a\}\{y\}})||(\langle\langle b,0\rangle\rangle_{\{b\}\{y\}};\langle\langle y,0\rangle\rangle_{\{b\}\{y\}})$

The confluent choice operator insists that the languages being combined are consistent. It is less obvious that the third language is consistent with the first two, since the value generated on y is different, but a simple case analysis shows that it is: only when the environment provides both $\langle a, 0\rangle$ and $\langle b, 0\rangle$ does the third language generate $\langle y, 0\rangle$. However, the first two languages do not accept this pattern and therefore do not generate any value on y, which is consistent.

Although only the third language explicitly says that events must be consumed on both a and b, confluent choice insists that exactly one event must be consumed on each channel before the whole block terminates.

5.2 Completeness

We now establish that confluent choice and sequential processes are expressive enough somehow.

Whatever I finite, O finite, $I \cup O$ non-empty, and V countable non-empty, the set T_{IO} is infinite and countable. Let t_0 be the empty trace. For all $n \in (\mathbb{N} \setminus \{0\})$, there exists a trace $t_n = \langle c_n^1, v_n^1\rangle \ldots \langle c_n^{|t_n|}, v_n^{|t_n|}\rangle \in T_{IO}$ such that $T_{IO} = \{t_n\}_{n\in\mathbb{N}}$.

Let L_0 be the $\mathscr{E}_{IO}$. For all $n \in (\mathbb{N} \setminus \{0\})$, let L_n be the language $\langle\langle c_n^1, v_n^1\rangle\rangle_{IO}; \ldots ; \langle\langle c_n^{|t_n|}, v_n^{|t_n|}\rangle\rangle_{IO}$ so that L_n is the language of all prefixes of t_n, thus the least confluent language containing t_n.

Theorem 9. *If $L \subseteq T_{IO}$ then there exists $S \subseteq \mathbb{N}$ such that $L = \{t_n\}_{n\in S}$. Moreover, if L is confluent then L is the confluent choice of the set of languages $\{L_n\}_{n\in S}$.*

Proof. If L is confluent and contains L_n for all $n \in S$ then their union is consistent and their confluent choice is well defined. Let it be L'. To start with $L' \subseteq L$ since L' is the least confluent language containing T_n for all $n \in S$. Reciprocally, for all $n \in S$, $t_n \in L_n \subseteq L'$. Hence, $L \subseteq L'$. $\square$

Every confluent language can be obtained as the confluent choice of a finite or infinite collection of sequential languages.

6 Conclusions

We have presented a characterization of confluent processes. When these processes are combined, they guarantee the overall behavior of the system—the number and sequence of values communicated on each channel—is the same regardless of local nondeterministic decisions made by each process. Such behavior can model, for example, the differences in execution time that comes from different implementations.

Our key definition is that of a confluent process, which behaves functionally: when placed in an environment characterized by the sequence of inputs it is willing to provide and the number of output events it is willing to accept, the process will generate a unique number and sequence of outputs. By insisting that this is a function, our processes adhere to the Kahn principle and guarantee overall system behavior.

Our main contribution is a series of operators that guarantee confluence and can be used to specify any confluent process. In addition to proving that classical sequencing, Kleene closure, and parallel composition preserve confluence, we introduce the confluent choice operator, which provides a way to deterministically merge both inputs and outputs and proves to be somehow fundamental for specifying processes that cannot otherwise be specified by Kahn's sequential language plus parallel composition.

Our goal in starting this work was to build the formal underpinnings of a flexible language for describing delay-insensitive hardware and software systems based on the Kahn principle, but removing some of its drawbacks such as undecidability and scheduling challenges. Our ongoing work involves building a user-level language around these primitives and incorporating them into a hardware/software codesign environment that avoids many usual problems in concurrent systems, such as races and nondeterminism brought on by shared variables.

At a more theoretical level, we have observed that confluent choice does not preserve regularity. Hence, we want to further analyze confluence w.r.t. regularity, so as to guarantee both confluence and regularity and specify processes both regular and confluent effectively.

References

[1] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[2] Gérard Berry and Ellen Sentovich. An implementation of constructive synchronous programs in POLIS. *Formal Methods in System Design*, 17(2):165–191, October 2000.

[3] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 37–44, Jersey City, New Jersey, September 2005.

[4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, New Jersey, 1985.

[5] Mark B. Josephs. Receptive process theory. *Acta Informatica*, 29(1):17–31, February 1992.

[6] Mark B. Josephs. An analysis of determinacy using a trace-theoretic model of asynchronous circuits. In *Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 121–130, Vancouver, BC, Canada, May 2003.

[7] Mark B. Josephs and Jan Tijmen Udding. An overview of D-I algebra. In *Proceedings of the 26th Hawaii International Conference on System Sciences*, volume I, pages 329–338, Hawaii, January 1993.

[8] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing*

74: Proceedings of IFIP Congress 74, pages 471–475, Stockholm, Sweden, August 1974. North-Holland.

[9] Paul G. Lucassen. *A Denotational Model and Composition Theorems for a Calculus of Delay-Insensitive Specifications*. PhD thesis, University of Groningen, May 1994.

[10] Nancy Lynch and Eugene Stark. A proof of the Kahn principle for Input/Output automata. *Information and Computation*, 82(1):81–92, July 1989.

[11] Sharad Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(7):950–956, July 1994.

[12] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[13] Robin Milner. *Communication and Concurrency*. Prentice Hall, Upper Saddle River, New Jersey, 1989.

[14] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, Berkeley, 1995. Available as UCB/ERL M95/105.

[15] Thomas Robert Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, University of California, Berkeley, October 1996. Memorandum UCB/ERL M96/76.

[16] Scott F. Smith and Amy E. Zwarico. Correct compilation of specifications to deterministic asynchronous circuits. *Formal Methods in System Design*, 7(3):155–226, November 1995.

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

[18] Kees van Berkel. *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*. Cambridge University Press, 1993.

[19] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Raeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proceedings of European Design Automation (EDAC)*, pages 384–389, Amsterdam, The Netherlands, February 1991.

[20] Kees van Berkel and Martin Rem. VLSI programming of asynchronous circuits for low power. In G. Birtwistle and A. Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 151–210. Springer-Verlag, 1995.

[21] Jan L. A. Van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.