Gates Accept Concurrent Behavior

Vineet Gupta
Vaughan Pratt[*]
Dept. of Computer Science
Stanford University, CA 94305
{vgupta,pratt}@cs.stanford.edu

Abstract

We represent concurrent processes as Boolean propositions or gates, cast in the role of acceptors of concurrent behavior. This properly extends other mainstream representations of concurrent behavior such as event structures, yet is defined more simply. It admits an intrinsic notion of duality that permits processes to be viewed as either schedules or automata. Its algebraic structure is essentially that of linear logic, with its morphisms being consequence-preserving renamings of propositions, and with its operations forming the core of a natural concurrent programming language.

1 Introduction

We understand sequential behavior very well today, namely in terms of acceptance of such behavior by sequential automata, e.g. a pushdown automaton accepting an English sentence. We propose to understand concurrent behavior via acceptance by concurrent automata of a particularly simple kind, namely Boolean gates as devices implementing Boolean propositions. Since the notion of a general Boolean gate, as just any n-ary operation $f : 2^n \to 2$, is simpler than the notion of say a pushdown automaton, we can expect the resulting mathematics of concurrent behavior to be correspondingly more elegant than that of sequential behavior. It is less obvious what aspects of concurrent behavior are expressible using gates. We shall show that gates as acceptors properly subsumes the event structure model [NPW81, Win88a], with the extensions being useful, while both simplifying it and improving its algebraic structure considerably.

Computational behavior is understood today in terms of formal languages as sets of possibly infinite strings, referred to in concurrency circles as *trace semantics*. Choice is expressed by a global choice of string from the language, made once at the beginning of the computation, while time is expressed by the linear order of symbol occurrences or *events*, also a global notion.

Branching time, first raised as an issue by Robin Milner [Mil80], makes choice a local phenomenon by distinguishing late branching $a(b + c)$ from early branching $ab + ac$ for atomic a and b, an issue that arises for nondeterministic programming languages. This distinction is needed because the program $ab + ac$ specifies a choice made before learning the outcome of the a, creating

[*]This work was supported by ONR under grant number N00014-92-J-1974, and a gift from Mitsubishi. The first author was also supported by an Accel Partners Fellowship.

the possibility of a deadlock that the better-informed former program $a(b+c)$ may be able to avoid. The resistance that met this notion in its first five years has since evaporated.

Automata respect the branching time distinction, but also draw many other less relevant distinctions, such as how many times a while loop has been unrolled. Thus one seeks a model intermediate in abstractness between automata and formal languages.

True concurrency makes time a local phenomenon by relaxing the linear ordering of events to a partial order and so relaxing the requirement that all spatially remote pairs of events have a well-defined temporal order. In this way true concurrency distinguishes parallel composition $a\|b$ from the choice $ab + ba$, even for atomic events a and b, by taking $a\|b$ to mean the performance of a and b *without regard for order* rather than in either order, a distinction that normally passes unnoticed in both everyday conversation and mathematics but not in treating mutual exclusion or *mutex*. Trace semantics in contrast obtains the meaning of $a\|b$ by decomposing a and b into their atomic constituents and identifying $a\|b$ with $ab + ba$ for atoms a and b.

This distinction was understood as an issue earlier than branching time, the earliest proponents including Petri [Pet62], Greif [Gre75], Mazurkiewicz [Maz77], Grabowski [Gra81], Nielsen, Plotkin, and Winskel [NPW81], and the second author [Pra82]. Yet the need for this distinction has proved even more controversial than the need for branching time. Here are our reasons for making this distinction.

First, the decomposability premise of trace semantics requires the process to be an actual implemented program as opposed to a mere specification, and furthermore that this implementation be visible to the verifier. The former assumption prevents reasoning about the behavior of processes for which only an abstract specification is available, while the latter ignores those vendors who for either commercial advantage or flexibility of maintenance keep their implementation hidden, in either case preventing customers from reasoning about their purchase if the above meaning of $a\|b$ is the only one available. Second, trace semantics assumes global time, contradicted by the physical reality of relativity and the engineering reality of delays in networks. Third, it assumes that atomicity is a fixed predicate, when in practice one finds the need to vary granularity, or *refine atomicity* as it is called in concurrency circles. In some contexts one regards a=b+c as one instruction, in others as two fetches, an addition, and a store, in yet others as the movement of many bits or even of many electrons.

We show in section 5 how the gates-as-acceptors view of concurrent behavior draws the distinction between true concurrency and mutex.

Besides the general contribution made by the overall gates-as-acceptors framework, we also make the following specific technical contributions.

• Schedules and automata constitute respectively declarative and imperative programming styles. We pair these up by representing both as binary relations, with the pairing defined in a strikingly simple way by converse (transpose) of binary relations. This pairing turns out to be a duality analogous to vector space duality in that it also pairs up transformations between gates, with each pair going in opposite directions.

• This duality is the negation operation of the Chu algebra of gates, constituting a previously studied model of Girard's linear logic [LS91, Bar91]. We define the remaining primitive operations of linear logic by the novel use of circuits, giving them a clear operational meaning in terms of behavior accepted by those circuits. And we further interpret these operations as useful connectives of a concurrent programming language.

• Noting that the representation of early branching and mutex as the respective regular expressions $ab + ac$ and $ab + ba$ requires mentioning a twice in each case, we shall represent these as gate-based schedules that mention each event only once. This eliminates the usual dependence on labeling needed to distinguish these two notions from respectively late branching $a(b + c)$ and true concurrency $a||b$.

In addition, by tying this framework into the algebra of Chu spaces [Bar79, LS91, Bar91, Pra93] we implicitly bring to bear on gates-as-acceptors the by-now-considerable machinery of both Chu-ology and linear logic, with the expectation moreover of seeing many further developments in this very interesting and (we conjecture) extremely rich new field.

2 Processes as Gates

2.1 Dynamic Acceptance

We propose to understand processes in terms of Boolean gates or propositions as acceptors of the behavior of their inputs. We define a gate $P = (A, X)$ to consist of a set A of inputs each monitoring one event, and a set $X \subseteq 2^A$ of subsets of A called states, constituting the satisfying assignments of the proposition. A subset of A denotes the assignment of the value 1 or true to its members and 0 or false to its nonmembers, and the *satisfying* assignments are those that make the output 1. (This definition remains sound for infinite A and X, required for infinite behaviors.)

By analogy with formal languages we might view each satisfying assignment as an accepted state, making $X \subseteq 2^A$ the analog of $L \subseteq \Sigma^*$. This static view is the wrong analogy however, since it contains no notion of dynamics: while there is global choice (the gate either accepts this assignment or that), there is no notion of time, global or otherwise.

Instead we interpret acceptance dynamically, regarding the inputs as toggling independently, asynchronously, and monotonically, namely from 0 to 1 (and hence toggling at most once), with the total such activity being accepted when the output remains constantly at 1 throughout. Intuitively this toggling activity constitutes a behavior. We associate each input with a distinct event, and identify the unique 0-1 transition of that input with the occurrence of its associated event.

We may depict $X \subseteq 2^A$ naturally as the Hasse diagram for 2^A ordered by inclusion, with the elements of X shown in black and the rest white, as illustrated by the two-event examples of Figure 1.

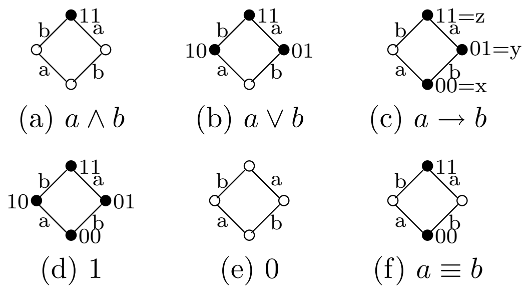


Figure 1. Six 2-input gates

We interpret these gates as representing the following processes. The conjunction $a \wedge b$ describes a process that starts with both inputs at 1, and which goes nowhere. The disjunction $a \vee b$ starts in a nondeterministically chosen state 10 or 01 and the zero input then toggles. The implication $a \rightarrow b$ toggles first a then b and hence is sequential. The constantly true gate $1[a, b]$ (our notation for indicating variables present yet not otherwise named in the proposition) permits a and b to toggle independently, denoting their pure noninteracting concurrence. The constant $0[a, b]$ is the inconsistent gate, which does not behave at all and cannot be observed. The equivalence $a \equiv b$ synchronizes its inputs. (Calibrate by comparing how you would have defined synchronization in your favorite model of concurrent behavior, bearing in mind that the model here has a delightful algebra.)

2.2 Behaviors accepted by gates

The restriction to monotone inputs limits behaviors to upward movement in the automaton, with respect to the orientation of Figure 1, putting us in the world of event-structure-like models, where the natural notion of behavior is an ascending *path* in the A-cube. Natural, that is, for events that happen sequentially. In the "true concurrency" hierarchy, a path or trace going up one event at a time is called *interleaving* semantics, while one going up several events at a time is called *step* semantics [DDNM88] (this term motivated by Petri net terminology [Rei85, p.20] and the form of concurrency implicit in SCCS [Mil83]), both being weaker [GG89] than *partial order* semantics [Gre75, Gra81, Pra82]. According to [Pra91, GJ92], a "truly concurrent" trajectory is a homotopy class of paths, meaning a broad but hole-free path, where a hole denotes mutual exclusion at the hole and hence implies a choice of which side of the hole to go round, i.e. which order the mutexed processes should go in.

However it is unreasonable in practice to insist that a particular behavior of a process contain no choices, since at any given level of granularity of behavior one may prefer to gloss over choices that are unimportant at that level. In the absence of any obvious structure that a run should always have, we define a *behavior* Y of a gate (A, X) to be a subset $Y \subseteq X$. Dually we define a *property* (or consequence) Y of (A, X) to be a superset $X \subseteq Y \subseteq 2^A$ of X.

(A, X) is uniquely determined by the set of all its behaviors, namely as their disjunction (union of all such Y's), as well as by the set of all its properties, namely as their conjunction (intersection of all such Y's).

For $X \subseteq Y$ we write $(A, X) \models (A, Y)$, calling (A, X) a behavior or model of (A, Y) and conversely (A, Y) a property or consequence of (A, X). Ordinarily $\models$ is a relation from structures to propositions, but gates as acceptors straddle this traditional boundary and are at home in either role, with $\models$ then being a transitive relation. Thus in figure 1 we have the chain $(e) \models (a) \models (f) \models (c) \models (d)$.

2.3 Dual views of a gate

A schedule of events is an intrinsically declarative program, while a state automaton is of course imperative. Figure 1 depicted gates as automata. Here we give a method of depicting a gate as the schedule dual to that automaton.

The idea is to work with the $A \times X$ membership relation defining $X \subseteq 2^A$, as an ordinary binary relation that we can transpose to form the dual gate (X, A), whose automaton we shall then regard as the schedule view of (A, X). The black points of the Hasse diagram will then denote the real events, those constituting A, and the dimensions of the diagram denote the states of X.

Applying this to Figure 1, we first obtain their binary relations as follows, with row a above row b.

$$\begin{array}{cccccc} 1 & 011 & 100 & 0101 & 2 \times 0 & 01 \\ 1 & 101 & 110 & 0011 & & 01 \\ & & & & & \\ (a) & (b) & (c) & (d) & (e) & (f) \end{array}$$

Figure 2. Fig 1. as binary relations

Now Figures 2 (a), (e), and (f) have repeated rows, whence their transposition is not the relation of any proposition, since X must be a set of states, not a multiset. We refer to as T_0 a proposition having no repeated rows, by analogy with topology, regarding A as the points of a topological space and the elements of X as its open sets. The T_0 propositions are equivalently those having among its properties no equivalences $a \equiv b$ save vacuous properties of the form $a \equiv a$. These are the propositions whose duals exist as propositions, whose automata we can therefore draw.

It will be seen from this construction that time in a schedule flows downwards. As a reminder of this, and to indicate that this diagram is intended as a schedule rather than an automaton, we attach a downward arrow to the upper left of each schedule, and if there is any ambiguity, an upward arrow to the lower right of each automaton, as in Figure 9.

The automata of the T_0 propositions of Figure 1 are as in Figure 3.

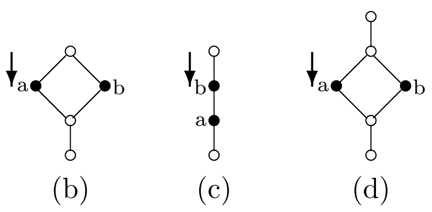


Figure 3. Schedule view of Figure 1

In addition to obtaining a dual Hasse diagram in this way, we also obtain a dual Boolean proposition, having states instead of events for its variables.

2.4 Causal structure

In all the pictures above, the structure of the gate is given by a boolean algebra $\mathbf{2}^A$ or $\mathbf{2}^X$. It is natural to ask if we can represent this by a simpler picture. We could just have a poset corresponding to 2^A, but we shall show that this is not enough. We do need almost the entire boolean algebra structure to represent the gate. For A infinite, such a Boolean algebra must be further specified to be complete (has arbitrary meets and joins) and atomic (every nonzero element dominates some atom), standardly called a *CABA*. We refer to the CABA that X is embedded in as equipping X with its *causal structure*, thereby turning X from a mere set of states to a *state space.*

The "almost" is prompted by the following theorem, valid for all cardinalities of A, that for finite A permits a savings of up to an exponential in depicting gates as two-toned Hasse diagrams as in Figure 1 (without the bit-string labels). Lattice theoretically this is a basic theorem about unique embedding of profinite distributive lattices [Joh82, p.250] in CABA's; here we give the construction in more elementary terms for greater accessibility.

Theorem 1 *Let $X \subseteq B$ generate*[1] *the CABA B, and let $L \subseteq B$ be any subcomplete lattice*[2] *of B such that $X \subseteq L$. Then B is the unique extension of L to a CABA generated by X.*

That is, given just the Hasse diagram of such a lattice L, with $X \subseteq L$ indicated, we can deduce not only the full n-cube from whence it came but where L must embed in it, i.e. what bit-strings to label both the real (X) and imaginary ($L - X$) states of L with.

Proof: The idea is to identify certain elements of L, which we then put in 1-1 correspondence with the atoms of B. We take A to be the set of sup-irreducible elements of L, definable as those elements not expressible as the sup (the arbitrary, i.e. possibly infinite or empty, join) of the set of elements strictly below them. We embed L in the CABA 2^A via $f : L \to 2^A$ defined by $f(y) = A \cap \downarrow y$ where $\downarrow y = \{x \in L \mid x \leq y\}$. This function can be seen to preserve all joins and meets of L and to be injective (argue along the lines of [MMT87, pg 83, thm 2.55]).

In order to show that A is the smallest set such that L embeds in 2^A, suppose L embeds in $2^{A'}$ by the embedding $g : L \to 2^{A'}$. Then for each sup-irreducible $x \in L$, let $a_x \in g(x) - \bigcup_{y<x} g(y)$. Such an a_x exists as x is sup-irreducible. Now for any two sup-irreducibles, if $a_x = a_y$ then $a_x \in (x \wedge y) < x$, which would contradict the definition of a_x. Thus there are as many elements in A' as in A. So 2^A is the smallest CABA containing L. However X generates B, and $X \subseteq L$, so B is also the smallest. Thus 2^A is isomorphic to B. We can now construct an isomorphism between 2^A and B such that L is embedded similarly in both, by mapping each $x \in A$ to the a_x constructed earlier. ∎

For example Figure 1(c) can be redrawn as a chain (a distributive lattice) with a reduction from $2^2 = 4$ elements to $2 + 1 = 3$ elements, more generally from 2^n to $n + 1$. The theorem applies only to T_0 gates, so it does not apply to Figure 1(a,e,f). It is evident that X generates 2^A iff (A, X) is T_0.

In drawing the state spaces of T_0 gates we therefore need draw only the lattice of those subsets of A generated by the accepted states under arbitrary (including empty) union and intersection. We call this the *partial distributive lattice* or PDLat presentation of a state space, where the lattice of subsets is the *underlying lattice*, and the accepted states are its *points*. This saves drawing the white state of Figure 1(c) for example.

One might ask whether we can recover the CABA B given even less than the PDLat structure on X. The partial order on X induced by its embedding in B does not suffice, the simplest counterexample for which is the pair of Boolean algebras in Figure 4(a-b), having the same poset X, namely $X = \emptyset$ (we do not insist on $0 \neq 1$ for Boolean algebras). Figure 4(c-d) gives a more informative counterexample (we have arbitrarily chosen to draw schedules, but the idea holds equally for automata).

If however we record not just partial order information but all properties of the form $a = b \vee c$ and $a = b \wedge c$ for $a, b, c \in A$, then we can distinguish (d) from (c) as satisfying $a \vee b = c$. But

[1]That is, B has no proper sub-CABA containing X

[2]That is, a "sub-(complete lattice)" of B, meaning a subset of B closed under arbitrary (including empty and infinite) meets and joins of B.

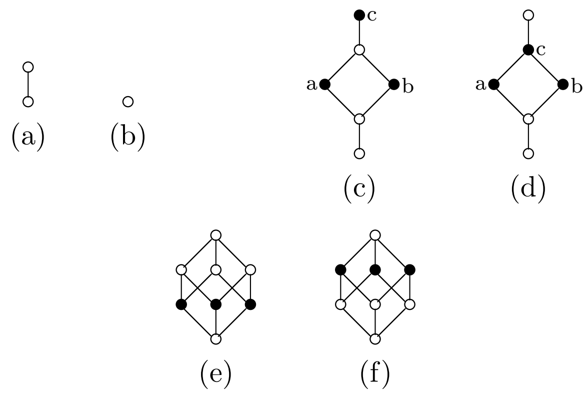


Figure 4. Incompleteness of Poset Structure

this is still not enough, as Figures 4(e-f) show; these satisfy no nontrivial such equations, yet are not isomorphic. The full causal structure however does suffice, since with it we can name every gate: here 4(e) is the symmetric function $(a \vee b \vee c) \wedge \neg(ab \vee bc \vee ca)$, and dually 2(f) is $\neg(abc) \wedge (a \vee b) \wedge (b \vee c) \wedge (c \vee a)$.

3 Modeling Behavior

3.1 Aspects of concurrency

Recall that a property of a gate is a superset of its state space, equivalently, a Boolean consequence of the gate viewed as a Boolean proposition. The following succinctly expressed properties correspond naturally to various aspects of concurrency. This section serves three purposes: to further illustrate the use of gates in characterizing concurrent behavior; to show how quite a variety of aspects of behavior all come out of the one uniform notion of Boolean gates as acceptors (in AI this would be called *emergent* characterization); and to point up certain aspects absent from most or all other models of concurrency, in particular the notion of causality formalized here; and to illustrate the use of event propositions such as $a \wedge b \equiv c$. Any property expressed using states is taken to be a property of the dual gate.

Transition: A state x can evolve into a state y just when $x \to y$. These are just inclusions between states, all of which we regard as legitimate state transitions.

Temporal order: a occurs before b if $b \to a$. Winskel writes this as $a \vdash b$, called prime enabling [Win88b]. If $b \to a$, then every state with $b = 1$ must also have $a = 1$, which means that a must have been set to 1 no later than b.

Enabling: General (nonprime) enabling is $a, b \vdash e$; $c, d \vdash e \equiv e \to (a \wedge b) \vee (c \wedge d)$ (either of (a with b) or c with d suffices to enable e. This can be extended to three or more pre-events $a, b, c \vdash e$ and three or more $\vdash$'s.

Conflict: $a\#b$ is $\neg(a \wedge b)$ (binary or coherent conflict[NPW81]), and means that it is illegal to set both a and b to 1, i.e. they are in conflict. More generally, $\#x$ may be defined as $(\bigwedge_{a \in x} a) \equiv 0$ (no state contains all events in x, i.e. no superset of x is a state of X.)

Internal choice: $x \equiv y \wedge z$ and $\#(y \vee z)$ expresses the choice of conflicting states y or z, made in the state x, so this choice is "internal". This corresponds to the branching construct of programming languages, where a choice is made based on the information accumulated in the current state. We can have a choice between several states, like a `case` statement in C.

Dilemma: $v \wedge w = 0$ and $\#(v \vee w)$ expresses no basis for choosing between v and w, since it means making the choice when no events have occurred, and thus no information is gathered.

Causality: $a \wedge b \equiv c$ asserts that a and b jointly cause c as their *immediate* effect, as it is impossible to have done both a and b without doing c also. On the other hand $c \rightarrow a \wedge b$ is mere prime enabling, $a, b \vdash c$, that is it is OK to wait for a while before doing c. This distinction is absent from all other models of concurrency we are aware of.

Nondeterminism: $a \equiv b \vee c$ and $b\#c$ asserts choice of conflicting events b or c. Since the choice is made at the same time as doing a, any information gathered by doing a could not have been used to choose, however, the choice was not available before a. So this choice is made by the environment, and is "external". This contrasts with $b \vee c \rightarrow a$, which is mere prime enabling $a \vdash b,\ a \vdash c$.

Synchronization: $a \equiv b$ asserts that a and b must happen simultaneously. A T_0 gate has only identity synchronizations $a \equiv a$. Conditional synchronization, $a \rightarrow b \equiv c$ however may hold even for T_0 gates: it holds for causality $a \wedge b \equiv c$.

Programming a process. We mentioned before that a gate is a conjunction of all its properties. This gives us a declarative way of programming a given process, namely, we can write small gates for each of the properties that we want the process to satisfy, and "and" them together to get the full process. An alternative way of programming a gate is by carving out its automaton. We start out with an n-dimensional automaton, where n is the number of events in the process. Then we look at each of the vertices and color them black or hollow according as whether they are desirable or undesirable. The resulting picture gives the desired automaton for the process.

We give an example for the first method of programming by constructing a gate for the mutex process, described in detail in the last section. It has four inputs designated σ, τ, a, b and there are four properties that these must satisfy: $\sigma\#\tau$, $a \rightarrow \sigma \vee b$, $b \rightarrow \tau \vee a$, and $a \vee b \rightarrow \sigma \vee \tau$. The gate on the left shows how to do this modularly, and the gate on the right shows the full circuit by substituting the actual circuits for the modules.

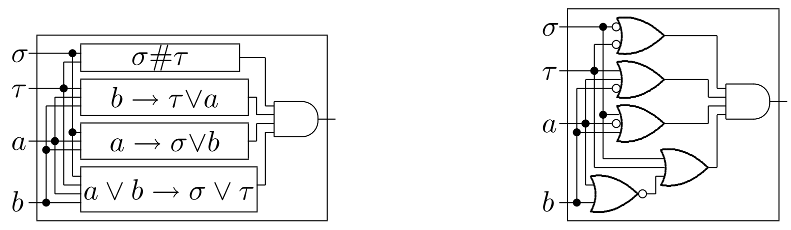

σ
τ
a
b
$\sigma\#\tau$
$b \rightarrow \tau \vee a$
$a \rightarrow \sigma \vee b$
$a \vee b \rightarrow \sigma \vee \tau$
σ
τ
a
b

Figure 5. Circuit for Mutual Exclusion

3.2 Comparison with Event Structures

In his survey paper on concurrency[Mil90], Robin Milner says

> The *event structures* introduced by Nielsen, Plotkin and Winskel [27] provide an important model in which these different views can be studied in the same framework;...

This section shows that gates as acceptors subsume the most general case of Winskel's event structures. As we have shown above, conflict and enabling of events can be expressed in a gate. Since these are the only two concepts required in the definition of an event structure [Win88b], we can show that event structures can be modeled by gates, for what amount to trivial reasons.

An event structure is a set of events E, with a conflict relation $\#$ and an enabling relation $\vdash$, defined in [Win88a]. A configuration is any subset of events which could have occurred in the event structure, so it must be conflict free, and every event must be caused by some previous events in the set according to the enabling relation. Given an event structure $E = (E, \#, \vdash)$, let $\mathcal{F}(E)$ be its set of configurations. We form the gate $G = (E, S)$, where $S = \mathcal{F}(E)$. Clearly these two are equivalent, and this leads us to the following theorem.

Theorem 2 *For every event structure that is formed from its family of configurations, there is a gate which has the same properties.*

If the event structure is labeled, we can define a labeling function on the gate by $\lambda : E \to \mathbf{Act}$, thus making this embedding respect labeling.

The domains that can arise as a family of configurations of an event structure have been characterized in [Dro89] as an algebraic complete partial order with some additional properties. It seems that the restrictions on these domains are placed to match their expressive power to that of Petri nets, and especially to outlaw causality. Gates however place no such restrictions, thus strictly subsuming event structures and allowing us to represent causality as well.

4 Algebraic Aspects of Gates

4.1 Chu Spaces

The simple notion of Boolean proposition is customarily algebraicized via Boolean algebras. This is however the wrong algebra for gates as acceptors. We instead treat gates as *Chu spaces*[3], a uniform generalization of sets and Boolean algebras, along with various semilattice and distributive lattice structures in between these two extremes, as well as their associated Stone duality, to be treated in detail in a forthcoming paper. As we have seen, propositions may also be understood as either binary relations or partial distributive lattices. In each of these perspectives their associated transformations are understood as respectively consequence-preserving renamings of Boolean propositions, "continuous functions" of binary relations (thinking of their rows as points of a topological space and their columns as its open sets), and homomorphisms of PDLat's. The category $\mathbf{Chu}(V, k)$ of Chu spaces and their associated transforms, with dualizing object k, was first defined in Po-Hsiang Chu's master's thesis, which appeared as an appendix to his advisor Michael Barr's monograph on *-autonomous categories [Bar79]. Here and in [Pra93] we follow Lafont and Streicher [LS91] in focusing on Chu's construction for $V = \mathbf{Set}$. In the present paper we further take $k = \{0, 1\}$.

Numeric quantities are not the only mathematical objects on which one can perform arithmetic. One may also add and multiply vector spaces, groups, etc. Chu spaces are no exception, and indeed

[3]Note that gates are extensional, but Chu spaces, being binary relations, are not required to be. Here we assume extensionality.

possess a very attractive algebra [Pra93], whose primitive operations can be taken to be sum $P+Q$ and its unit 0, tensor product $P \otimes Q$ and its unit $\top$, and dualization or perp $P^{\perp}$. To this one may add Girard's "exponential" $!P$ to complete the language to exactly that of linear logic [Gir87].

The present section treats this algebra from our gates perspective, in particular showing that all these operations and constants except $P^{\perp}$ can be defined using ordinary Boolean circuits, in a way that should make their meaning at least as clear, to those used to deciphering circuits, as any symbolic definition. The next section shows how these operations can be used in a concurrent programming language, providing motivation for them independent of their use in linear logic, whose proof theoretic motivation remains largely obscure to those not in the inner sanctum of proof theory.

As usual, certain composites of the primitive operations have a sufficiently natural meaning to warrant their own notation, which can be introduced in one paragraph and ignored thereafter, as here. The De Morgan dual of sum is product $P \times Q = (P^{\perp} + Q^{\perp})^{\perp}$, with unit $1 = 0^{\perp}$, while that of tensor product is Girard's par, $P \oplus Q = (P^{\perp} \otimes Q^{\perp})^{\perp}$ (Girard writes $P \invamp Q$, the subject of some controversy), with unit $\perp = \top^{\perp}$. Linear implication $P \multimap Q$ is $P^{\perp} \oplus Q$, satisfying $(P \otimes Q) \multimap R \cong A \multimap (Q \multimap R)$, while "intuitionistic" implication $P \Rightarrow Q$ is $!P \multimap Q$, satisfying $(P \times Q) \Rightarrow R \cong P \Rightarrow (Q \Rightarrow R)$ (the need for isomorphism $P \cong Q$ in place of identity $P = Q$ will be explained shortly).

As we have described elsewhere [Pra93], at the syntactic level this algebra of Chu spaces looks just like the Peirce-Schröder-Tarski-Jónsson calculus of binary relations [Pei33, JT52], to within choice of symbology, with $P \times Q$ written PQ (intersection of relations) and $P \otimes Q$ as $P;Q$ for relational composition, etc. The calculi start to differ a little at the equational level: the Peirce calculus' version of tensor product is not commutative (calling for two implications or *residuals* P/Q and $P \backslash Q$ in place of $P \multimap Q$), intersection of relations as its direct product distributes over their union as its sum, sum and product are idempotent, and equations are actual identities rather than mere isomorphisms. Otherwise however these two equational logics look quite similar.

The difference is greatest at the semantic level. Union $P+Q$ in the Peirce calculus takes two $m \times n$ relations and yields an $m \times n$ relation, whereas in the Chu calculus the sum of an $A \times X$ relation and a $B \times Y$ relation is an $(A+B) \times (X \times Y)$ relation. Likewise relational composition in the Peirce calculus takes an $m \times n$ relation and an $n \times p$ relation and returns an $m \times p$ relation, whereas for Chu, tensor product takes an $A \times X$ relation and a $B \times Y$ relation and returns an $(A \times B) \times Z$ relation where Z may be regarded as the set of all bilinear forms on $A \times B$ (cf. Halmos: "The *tensor product* $\mathcal{U} \otimes \mathcal{V}$ of two finite-dimensional vector spaces $\mathcal{U}$ and $\mathcal{V}$ is the dual of the vector space of all bilinear forms on $\mathcal{U} \oplus \mathcal{V}$ [Hal74, p.40]).

We have already defined the operation $P^{\perp}$, as converse of a Chu space (in its relational form). In this section we shall define the remaining three primitive operations and two constants of Chu algebra in terms of circuits.

The units for sum and tensor product, namely 0 and $\top$, and the exponential $!P$, are all defined by circuits whose output is constantly 1 and hence whose inputs are ignored. Zero has no inputs, the tensor unit $\top$ has one input, and $!P$ has for its inputs those of P. (This definition of $!P$ satisfies the Girard axioms but also satisfies $!!P \cong !P$, leaving open the possibility of a less constrained alternative Chu interpretation for $!P$.)

The sum $P+Q$ is implemented as a circuit with components P and Q, by forming the conjunction of the outputs of P and Q, with the set of inputs of $P+Q$ then being the disjoint union $A+B$ of those of P and Q. This construction is illustrated in Figure 6, for which $|A| = 3$ and $|B| = 2$.

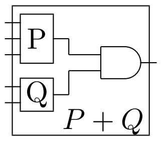


Figure 6. Circuit for Sum

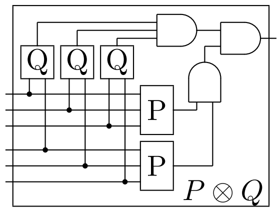


Figure 7. Circuit for Tensor Product

The gate thus implemented can be seen to partition its inputs into two disjoint blocks, one judged by P, the other by Q, with the gate as a whole registering its approval just when all its components approve of their respective blocks.

Tensor product $P \otimes Q$ is the only operation requiring some work, and is where the circuit approach to defining operations really helps. As one might guess from the fact that the tensor unit has output 1, tensor product is a form of conjunction. But whereas sum is a noninteracting conjunction, tensor product behaves like a generic logical inference, in which the constraints in the components can entail new constraints involving the variables of both components.

Now as with sum, tensor product assumes no *a priori* relationship between the inputs of its arguments, which are therefore made disjoint. Nevertheless a connection is established between the two sets of inputs, namely by taking the set of inputs of $P \otimes Q$ to be the cartesian product $A \times B$ of those of P and Q instead of their sum. Visualizing this product as a rectangular array of inputs, we define *bilinearity* to be the condition that each column of this rectangle (what one obtains by fixing a particular $b \in B$ of Q's inputs) independently satisfies P while each row satisfies Q. This is realized by using $|B|$ distinct copies of P to monitor each column of $A \times B$, and likewise $|A|$ copies of Q to monitor rows, as illustrated in Figure 7 for the case where the rectangle the 3×2.

4.2 Maps

We have mentioned isomorphisms of Chu spaces without defining them. More generally there exists a natural notion of morphism for Chu spaces, without which one could not distinguish Chu spaces from other families of subsets such as hypergraphs, which transform quite differently. With it, the *class* of Chu spaces becomes the *category* of such.

For the most general definition of morphism for Chu spaces, valid in any symmetric closed category V and defining the category $\mathbf{Chu}(V, \perp)$ for an arbitrary choice of $\perp$ as an object of V, see Chu's appendix to Barr's monograph [Bar79]. But the restriction to $V = \mathbf{Set}$, as assumed by Lafont and Streicher [LS91], appears to be good enough for ordinary applications, an intuition that we have substantiated with the theorem that the category of all n-ary relational structures and their homomorphisms fully and concretely embeds in $\mathbf{Chu}(\mathbf{Set}, 2^n)$ [Pra93, §5]. (Thus $\mathbf{Chu}(\mathbf{Set}, 2)$, the subject of this paper, fully extends the category of unary relations, where "full" means that the (necessarily faithful) embedding functor is full, i.e. the embedded objects transform in the "same" way.)

A *Chu transform*[4] from (A, X) to (B, Y) is defined to consist of a pair of functions $f : A \to B$ and $g : Y \to X$ satisfying $a \in g(y) \leftrightarrow f(a) \in y$ for all $a \in A$ and $y \in Y$. In the framework of gates such a transform can be expressed as an attachment of the inputs of P to the inputs of

[4]This definition works for extensional Chu spaces only, but can be generalized to all Chu spaces.

Q, specified by f in the evident way, having the property that every input accepted by Q is also accepted by P (the above equation). g constitutes an explicit function from inputs accepted by X to the corresponding ones accepted by P, and is completely determined by f here. (Thinking of accepted inputs as corresponding to open sets of a topological space whose points are the inputs, this is exactly the definition of continuous function: the inverse of f, here essentially g, must send every open set of X to some open set of P.)

A *renaming* is a function $f : A \to B$ from the set A of variables of P to the variables of Q; for example there are four renamings from $a \vee b$ to $c \oplus d$, namely $c \vee d$, $d \vee c$, $c \vee c$, and $d \vee d$. A renaming from P to Q is *consequence-preserving* when P so renamed is a consequence of Q, equivalent to requiring that the renaming take consequences of P to consequences of Q. Only the first two of the above four renamings are consequence-preserving.

Theorem 3 *The states of $P \otimes Q$ are in 1-1 correspondence with the Chu transforms from P to $Q^\perp$.*

Corollary *$P \multimap Q$, defined as $(P \otimes Q^\perp)^\perp$, is the Chu space of all Chu transforms from P to Q.*

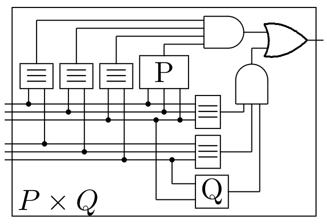
A PDLat homomorphism $f : P \to Q$ is a complete lattice homomorphism between their underlying lattices, which takes the points of P to the points of Q.

Theorem 4 *For T_0 propositions, extensional T_0 Chu spaces, and extensional PDLat's, the respective categories defined by Chu transforms, consequence preserving renamings and PDLat homomorphisms are equivalent.*

5 Process Algebra

We now assign operational interpretations to the Chu algebra operations, supplemented with an additional choice operation $P \sqcup Q$ that does not fit naturally into Chu algebra. We also take this opportunity to mention a circuit implementation of product.

The *concurrence* of two gates P and Q is their sum $P + Q$. Any behavior accepted by $P + Q$ is a concurrent non-communicating execution of a behavior accepted by P and a behavior accepted by Q. From its circuit, it is clear that the inputs to P and Q can be varied independently, representing concurrent execution.



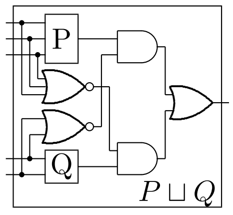


Figure 8. Circuits for product and choice.

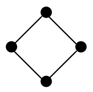
The operation $P \times Q$ may be defined in terms of $P^\perp$ and $P + Q$ as $(P^\perp + Q^\perp)^\perp$, making $\times$ the De Morgan dual of $+$. This operation turns out to be circuit-definable as in Figure 8, whose $\equiv$ modules output 1 when their inputs are all equal.

The flow or *orthocurrence* [Pra86, CCMP91] of two gates is $P \otimes Q$. The behaviors accepted are the interaction of two behaviors accepted by P and Q. For example, if three trains are running in the same direction on the same track, and pass two stations, there are six events, one for each train crossing each station. Each train will see the two stations pass in a certain order, and similarly each station will see the trains pass in a certain order. So a behavior for the whole scenario is one in which each of these local constraints are met. The bilinear nature of $P \otimes Q$ is expressed here by having one station-checker Q on each train and one train-checker P on each station and ensuring that all are satisfied.

The *choice* of P and Q, denoted by $P \sqcup Q$, is the gate which accepts any behavior which is either a behavior in P or a behavior in Q, such that events from both P and Q are not executed in the same behavior. The above circuit for choice implements this by making sure that no events in Q occur if P has started execution and vice versa.

6 Branching Time and True Concurrency

We now apply this notion of gate transformation to show how the relationships between early and late branching, and between true concurrency and mutex, can be handled in our framework. In more conventional frameworks this relationship cannot be treated in full without the assistance of labels to indicate which events are repeated instances of the one event. Early branching $ab + ac$ in its usual representations, including the regular expression $ab + ac$ itself, contains a repetition of a, while representations of mutex $ab + ba$ typically repeat both a and b.

Elsewhere [Pra91] we have used homotopy to give a label-free method of distinguishing true concurrency from mutex by representing the former as the usual product automaton  filled in as a solid square, and the latter as the same but with the interior hollow. In both cases this relies on the geometry of the situation to give the information that had we labeled the edges, parallel edges would have received the same label.

Here we do not have higher-dimensional cells to represent this information. Instead we use silent transitions, of the kind introduced by Milner [Mil80], for both early branching and mutual exclusion.

Branching Time. Late branching $a(b + c)$ is easily represented as the conjunction of $b \to a$, $c \to a$, and $b\#c$ $(b \wedge c = 0)$, that is, a precedes both b and c which cannot both happen.

To change this to early branching $ab + ac$ we adjoin to the late branching formula the further conditions $b \to \sigma$, $c \to \tau$, $\sigma\#\tau$ (making $b\#c$ redundant), and $a \;\to\; \sigma \vee \tau$ (the choice of σ or τ must be made before a is done), where σ and τ are silent transitions, whose only purpose is to commit to a choice, preceding respectively b and c.

Late and early branching may be depicted as per the respective schedules of Figure 9, or by their corresponding automaton, shown below. (Without Theorem 2 the schedule for early branching would have 256 events instead of just 17.)

The relationship between the late branching schedule and the early one is that of subspace: the former is a subspace of the latter, namely the circled elements in the schedule for the latter, on the right. For automata we have the dual relationship: the late branching automaton is a quotient of the early branching automaton, namely the dimensions σ and τ, shown dotted, are projected out, thereby identifying states t, u, v, and w, x.

The timing of this branching is only early relative to the process at hand, in that when combined with other processes the choice of σ or τ (and hence of b or c) need only be made prior to a, not at the beginning of the containing process. This can however be turned into absolute early branching by requiring that every disjunction of silent choices such as $\sigma \vee \tau$ precede every nonsilent action, forcing all choices to be made before any real activity commences. However there is a slicker way: simply require $\sigma \vee \tau = 1$. This rules out states in which neither σ nor τ have happened, thus forcing the containing process, however large, to start in a state that has already made this choice.

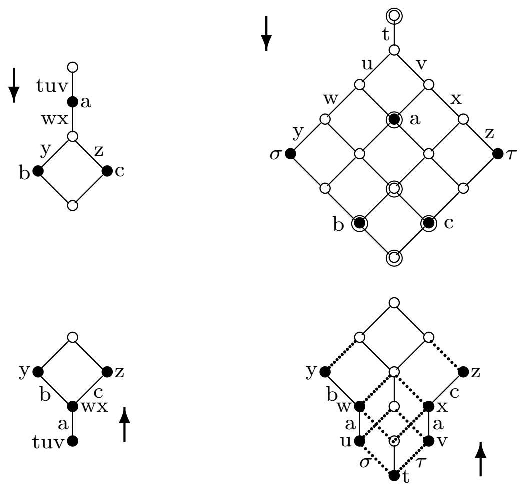


Figure 9. Schedules and automata for branching

The opposite of this is to insist on the choice being made in parallel with a, easily accomplished by strengthening $a \to \sigma \vee \tau$ to $a \equiv \sigma \vee \tau$. This is of course inconsistent with requiring $\sigma \vee \tau = 1$, in that it immediately entails $a = 1$. We leave as an exercise to calculate the diagrams for these variations on the basic theme.

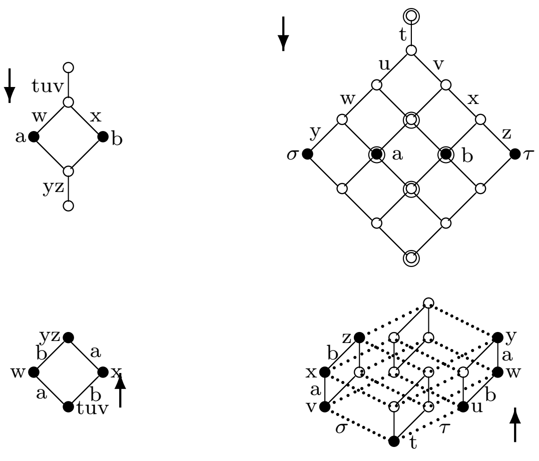


Figure 10. Schedules and automata for concurrency

True Concurrency, $a \| b$, is simply $1[a, b]$, the constantly true formula with inputs a and b. As with early branching, we shall represent the mutex $ab + ba$ with the help of silent transitions σ and τ

preceding a and b respectively. The conditions are $\sigma \# \tau$ (choice), $a \rightarrow b \vee \sigma$ (if σ is not chosen then b must precede a), $b \rightarrow a \vee \tau$ (conversely for τ), and $a \vee b \rightarrow \sigma \vee \tau$ (the choice must be made before doing either of a or b). The schedules and automata for true concurrency and mutual exclusion are as per Figure 10.

Again there is an inclusion from left to right sending a, b to themselves, with a matching projection from the seven states of the exclusive process to the four states of the concurrent one. And again we have the option of making the choice at any time prior to $a \vee b$, or at the beginning of time, representable for the mutex schedule by retracting the top spike.

What is remarkable is the great similarity of branching time and true concurrency as revealed by this analysis. On the right they have the same seven states, whereas on the left late branching identifies w and x while true concurrency instead identifies y and z.

References

[Bar79] M. Barr. *$*$-Autonomous categories, LNM 752.* Springer-Verlag, 1979.

[Bar91] M. Barr. $*$-Autonomous categories and linear logic. *Math Structures in Comp. Sci.*, 1(2), 1991.

[CCMP91] R.T Casley, R.F. Crew, J. Meseguer, and V.R. Pratt. Temporal structures. *Math. Structures in Comp. Sci.*, 1(2):179–213, July 1991.

[DDNM88] P. Degano, R. De Nicola, and U. Montanari. A distributed operational semantics for CCS based on condition/event systems. *Acta Informatica*, 26(1/2):59–91, October 1988.

[Dro89] M. Droste. Event structures and domains. *Theoretical Computer Science*, 68:37–47, 1989.

[GG89] R.J. van Glabbeek and U. Goltz. Equivalence notions for concurrent systems and refinement of actions. In A. Kreczmar and G. Mirkowska, editors, *Proc. Conf. on Mathematical Foundations of Computer Science*, volume 379 of *Lecture Notes in Computer Science*, pages 237–248. Springer-Verlag, 1989.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GJ92] E. Goubault and T.P. Jensen. Homology of higher dimensional automata. In *Proc. of CONCUR'92, LNCS 630*, pages 254–268, Stonybrook, New York, August 1992. Springer-Verlag.

[Gra81] J. Grabowski. On partial languages. *Fundamenta Informaticae*, IV.2:427–498, 1981.

[Gre75] I. Greif. *Semantics of Communicating Parallel Processes.* PhD thesis, Project MAC report TR-154, MIT, 1975.

[Hal74] P.R. Halmos. *Finite-Dimensional Vector Spaces.* Springer-Verlag, 1974.

[Joh82] P.T. Johnstone. *Stone Spaces.* Cambridge University Press, 1982.

[JT52] B. Jónsson and A. Tarski. Boolean algebras with operators. Part II. *Amer. J. Math.*, 74:127–162, 1952.

[LS91] Y. Lafont and T. Streicher. Games semantics for linear logic. In *Proc. 6th Annual IEEE Symp. on Logic in Computer Science*, pages 43–49, Amsterdam, July 1991.

[Maz77] A. Mazurkiewicz. Concurrent program schemas and their interpretation. In *Proc. Aarhus Workshop on Verification of Parallel Programs*, 1977.

[Mil80] R. Milner. *A Calculus of Communicating Systems, LNCS 92.* Springer-Verlag, 1980.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Mil90] R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 19, pages 1201–1242. Elsevier Science Publishers B.V. (North-Holland), 1990.

[MMT87] R. McKenzie, G. McNulty, and W. Taylor. *Algebras, Lattices, Varieties, Volume I.* Wadsworth & Brooks/Cole, Monterey, CA, 1987.

[NPW81] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures, and domains, part I. *Theoretical Computer Science*, 13, 1981.

[Pei33] C.S. Peirce. Description of a notation for the logic of relatives, resulting from an amplification of the conceptions of Boole's calculus of logic. In *Collected Papers of Charles Sanders Peirce. III. Exact Logic.* Harvard University Press, 1933.

[Pet62] C.A. Petri. Fundamentals of a theory of asynchronous information flow. In *Proc. IFIP Congress 62*, pages 386–390, Munich, 1962. North-Holland, Amsterdam.

[Pra82] V.R. Pratt. On the composition of processes. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, January 1982.

[Pra86] V.R. Pratt. Modeling concurrency with partial orders. *Int. J. of Parallel Programming*, 15(1):33–71, February 1986.

[Pra91] V.R. Pratt. Modeling concurrency with geometry. In *Proc. 18th Ann. ACM Symposium on Principles of Programming Languages*, pages 311–322, January 1991.

[Pra93] V.R. Pratt. The second calculus of binary relations. In *Proceedings of MFCS'93*, Gdańsk, Poland, 1993. Springer-Verlag.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* Springer-Verlag, 1985.

[Win88a] G. Winskel. A category of labelled Petri nets and compositional proof system. In *Proc. 3rd Annual Symposium on Logic in Computer Science*, Edinburgh, 1988. Computer Society Press.

[Win88b] G. Winskel. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, REX'88, LNCS 354*, Noordwijkerhout, June 1988. Springer-Verlag.