Math. Struct. in Comp. Science (2003), *vol.* 13, *pp.* 485–529. © 2003 Cambridge University Press
DOI: 10.1017/S0960129503004031 Printed in the United Kingdom

Transition and cancellation in concurrency and branching time

VAUGHAN R. PRATT

Stanford University

Received 8 March 2001; revised 11 June 2002

We review the conceptual development of (true) concurrency and branching time starting from Petri nets and proceeding *via* Mazurkiewicz traces, pomsets, bisimulation, and event structures up to higher dimensional automata (HDAs), whose acyclic case may be identified with triadic event structures and triadic Chu spaces. Acyclic HDAs may be understood as extending the two truth values of Boolean logic with a third value $\lrcorner\!\ulcorner$ expressing *transition*. We prove the necessity of such a third value under mild assumptions about the nature of observable events, and show that the expansion of any complete Boolean basis L to $L_{\lrcorner\!\ulcorner}$ with a third literal $\widehat{a}$ expressing $a = \lrcorner\!\ulcorner$ forms an expressively complete basis for the representation of acyclic HDAs. The main contribution is a new value $\times$ of *cancellation*, which is a sibling of $\lrcorner\!\ulcorner$, serving to distinguish $a(b + c)$ from $ab + ac$ while simplifying the extensional definitions of termination $\checkmark\mathscr{A}$ and sequence $\mathscr{A}\mathscr{B}$. We show that every HDAX (acyclic HDA with $\times$) is representable in the expansion of $L_{\lrcorner\!\ulcorner}$ to $L_{\lrcorner\!\ulcorner\times}$ with a fourth literal $\not{a}$ expressing $a = \times$.

1. Introduction

1.1. *Sequential and concurrent behaviour*

What distinguishes sequential computation, or for that matter sequential behaviour of any kind, from concurrent? The usual viewpoint draws the following temporal distinction:

Sequential behaviour allows only one event to happen at a time. With concurrent behaviour multiple events may occur simultaneously.

Implicit in this distinction is the traditional view of time as evolving steadily and independently, providing a background against which to observe the processing of information. This view makes the behaviour of time itself sequential – no two nanoseconds can overlap – while allowing arbitrarily independent behaviour for bits.

The point of view espoused in this paper views time and information more symmetrically as two complementary or dual spaces. We view the points of time not so much as temporal instants but as instantaneous events serving as state deltas: an event changes information but not time. Dually the points of information space are seen as time deltas: during a given state time passes while information remains fixed. These are, of course, idealised events and states; physical events always take time, even if only 10^{-20} seconds, while physical states are never completely stationary.

While this more symmetric view of time and information favours neither sequential nor concurrent behaviour, it does raise the question of whether they can be distinguished in

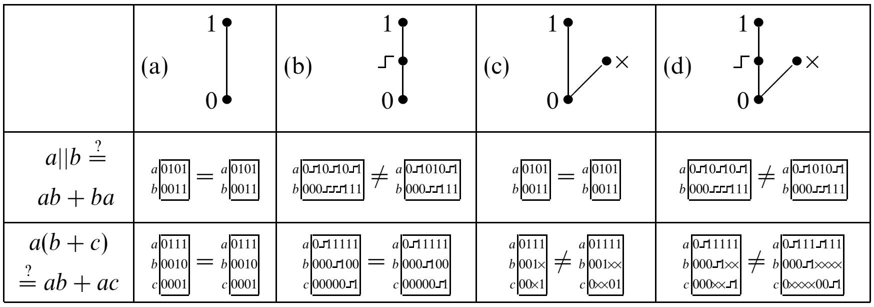

(a)
(b)
(c)
(d)
1
0
×
$a||b \overset{?}{=} ab + ba$
$a(b+c) \overset{?}{=} ab + ac$
=
≠

Fig. 1. Refinements of atomic time.

a symmetric way. We offer the following symmetric distinction:

Sequential behaviour synchronises the evolution of time and information. With concurrent behaviour time and information evolve more independently.

In particular, the sequential events (transitions) and states of traditional automata theory alternate in lock step, allowing time to be defined abstractly as the number of state changes. Concurrency decouples this relationship, with the result that the passage of both time and information can then only be measured in terms of each other in a distributed way. With the representation adopted below of processes as matrices over a set K, time and information are obtained naturally by standardly lifting a suitable generalised metric on K to rows and columns, respectively, *in the same way* to yield metrics on the respective temporal and information spaces. We shall treat this further in a more categorically-oriented follow-up paper for CONCUR'02, which focuses on enrichment.

1.2. *Refinements of atomic time*

The simplest non-trivial view of atomic time, the view of time from the perspective of an atomic event, is that of Figure 1(a). Time divides into just two regions or *values*, 0 and 1, according to whether the event has not or has happened, respectively. The event may pass from 0 to 1 (the figure orients time as flowing upwards), but having done so it may not return. That is, the (truth) value of the atomic proposition P_a expressing 'a has happened' can only increase monotonically with time. We write K for the allowed values, so in Figure 1(a) $K = \{0, 1\}$.

The process $a||b$ (perform events a and b independently) permits the states of a and b to be any of the four combinations 00, 10, 01, or 11, which appear as the four columns in the first box of Figure 1. The process $ab + ba$ (perform a and b in either order) permits the same states, so this choice of K satisfies $a||b = ab + ba$. The process $a(b + c)$ (perform a and then one of b or c) permits the states 000, 100, 110 and 101 for a, b, c, as does $ab + ac$ (perform one of a then b or a then c), so this K also satisfies $a(b + c) = ab + ac$.

Figure 1(b) refines this view with the recognition of an intermediate state ⎍ of *transition*. K now has 3 states, whence $a||b$ has $3^2 = 9$ states. $ab + ba$ has the same states with the exception of the state ⎍⎍ denoting a and b simultaneously in transition, and so ⎍ now distinguishes $a||b$ from $ab+ba$. But even though ⌐ also furnishes $a(b+c)$ and $ab+ac$ with additional states, the new states are the same for both, which therefore remain indistinguishable. Other roles for ⌐ include distinguishing asymmetric from symmetric conflict, representation of resource limits, and a satisfactory notion of running time for a parallel computation.

Figure 1(c) proposes a different way of adding a third state, namely the state $\times$ of *cancellation*. A choice may necessitate cancelling an event, for example, choosing a in $a+b$ automatically cancels b immediately, even before a happens. No choice is made in $a||b$, so there are no cancellations. But even though $ab+ba$ entails a choice, both a and b happen on both branches and so neither is cancelled in either case, leaving $a||b = ab+ba$.

Now in $a(b+c)$, after a happens (state 100) a choice is required, at which point one of b or c is cancelled (state $1{\times}0$ or $10{\times}$, respectively). In $ab+ac$ this choice is made at the beginning, at which time one of b or c is cancelled (state $0{\times}0$ or $00{\times}$, respectively, with state 100 now disallowed). So $\times$ distinguishes $a(b+c)$ from $ab+ac$. Other roles for $\times$ include a more satisfactory notion of final state, which leads to improved definitions of sequence and termination.

Figure 1(d) is the evident amalgamation of 1(b) and 1(c) (though, since the relationship of 1(a) to 1(b) is embedding but to 1(c) it is quotient, it would have to be some sort of mixed-variance amalgamation or epi-mono square construction). Tables 1–4, encountered along the way as our more detailed story unfolds, correspond to Figures 1(a)–(d), respectively.

We associate with the states 0, ⌐, 1, $\times$ of K the literals $\overline{a}$, $\widehat{a}$, a and $\cancel{a}$, respectively, asserting $a = 0$, $a =$ ⌐, $a = 1$ and $a = \times$, respectively. We shall show that every subset of K^A is representable as a disjunction of conjunctions of these four literals.

The earliest explicit representation of non-performance that we are aware of is Genrich and Thiagarajan's notion of L-value for Petri nets (Genrich and Thiagarajan 1984). Much more recently, and closer to our notion of cancellation, Rodriguez and Anger (2001) gives an analogous account of branching time in terms of an expansion of Allen's interval algebra (Allen 1984) of the 13 possible relationships between two intervals sliding past each other. The traditional account of interval algebra derives the 13 relationships from three binary relations $<$, $=$ and $>$, each between two endpoints, with one from each interval. Rodriguez and Anger consider a fourth relationship, $||$, in which the two endpoints are no longer comparable, as when the parallel tracks on which the intervals slide diverge at some point, deriving 19 interval relationships. Section 3.7 considers this notion of branching time in more detail, addressing the meaning of the symmetric relationship $||$ when reinterpreted as a value of K. We show that it would have the meaning of an event that blocks as opposed to being cancelled, with the result that using $||$ analogously to $\times$ to distinguish $a(b+c)$ from $ab+ac$ would cause d to block in $a(b+c)d$, a crucial distinction between the respective characterisations of branching time by $||$ and $\times$.

1.3. *Rationale*

For state-of-the-art software and hardware verification, the traditional state-oriented view of computation is all that is needed. In this view, a computer system, implemented in hardware, software or some mix of the two, is understood abstractly as an edge-labelled graph $G = (V, E)$. Each vertex $u \in V$ denotes a possible *state* of the system. Each edge $e \in E$ from u to v is labelled with a symbol or *action* $\lambda(e) = \mathbf{a}$ indicating that if the system is in state u and action $\mathbf{a} = \lambda(e)$ occurs, the next state of the system will be v (or *can* be v in the case of non-deterministic computation).

Correctness of the system so modelled is defined in terms of predicates at program points L that are required to hold for the system state x whenever the program counter (whether real or virtual) in x is at L. Only states, as the effects of transitions, are observable, not the transitions themselves.

The canonical application of this viewpoint is verification, whose function originally was conceived as an *imprimatur* certifying the whole program correct once and for all, but which in recent years has retargetted logical laws to serve as in-laws perpetually finding fault with the system, rooting out bugs in parallel with the evolution of the system design, while exposing tacit assumptions of the system designers. This view serves its intended purposes well, and has become the basis for a thriving (albeit still cottage) verification industry.

There are, however, two assumptions tying the system designer's hands here, namely fixed granularity and sequential observation.

Fixed granularity. This is the assumption that actions of the system are built up from atomic actions whose start and end are both observable, but not the period in between – the transition is quicker than the eye. This assumption justifies the equation $a||b = ab + ba$ identifying independent occurrence of atoms a, b with their occurrence in either order. A natural name for the latter is atomic mutual exclusion.

The assumption of fixed granularity is called into question by the widespread practice of maintaining confidentiality of source code. Confidentiality has two benefits, a competitive one of trade secrecy and a technological one of being able to improve the implementation without compromising the specification. However, it also has the effect of presenting some system components as atomic to the user even though the implementor sees them as built up from smaller atoms. While it certainly *suffices* for system correctness to organise such user-atomic components so as to satisfy atomic mutual exclusion, this requirement can in many cases impose a sufficient burden on both implementors and users as to justify asking whether the user *must* assume atomic mutual exclusion in order to verify larger systems built from such components. The study of concurrent computation bifurcates according to the answer – this paper is firmly in the camp dedicated to exploring what is possible without the assumption of fixed granularity, and hence without assuming atomic mutual exclusion.

Sequential observation. This is the assumption that every dispute can be resolved by one referee. From a legal standpoint, it might seem as though the nationals of any country with a Supreme Court would be on safe ground with this assumption, at least within their own borders, since that court will ultimately resolve any dispute of sufficient significance. But the resolution may come too late to be useful, or the one referee may be overwhelmed with such requests, or the centralised arbiter may recognise a prior period of time where one distributed system observed another with results that the arbiter can see in hindsight could not have been obtained at the time by a single observer even in principle, regardless of available resources (Plotkin and Pratt 1996).

The appropriate operator combining two distributed systems with one observing the other is orthocurrence (Pratt 1985; Pratt 1986; Casley *et al.* 1991; Gupta and Pratt 1993). Orthocurrence can be understood either symmetrically as the *interaction* $\mathscr{A} \otimes \mathscr{B}$ of two processes $\mathscr{A}$ and $\mathscr{B}$, or, ostensibly, asymmetrically as the *observation* $\mathscr{A} \multimap \mathscr{B}^{\perp}$ of states of process $\mathscr{B}$ from vantage points of process $\mathscr{A}$, or the dual observation $\mathscr{B} \multimap \mathscr{A}^{\perp}$ (giving a sense in which observation is really symmetric) (Pratt 2001). Besides being the only viable candidate proposed to date for this role, it has the additional benefits of an intuitive definition in terms of crossword puzzles, and attractive algebraic and logical properties, specifically, both k-2 logic (corresponding to k-adic Chu spaces) and Girard's linear logic (Girard 1987).

Now, orthocurrence makes no sense for the standard state-based or transition-system view of computation because its events consist of pairs of events each taken from one of two interacting or 'orthocurrent' systems. Product does appear in automata theory, but only for concurrence, which forms the product of state sets; in contrast, orthocurrence multiplies event sets. The incompatibility of orthocurrence with the standard state-based view of behaviour makes it of predictably limited interest for systems designed from that perspective.

We maintain, nonetheless, that orthocurrence as the multiplicative form of concurrency is every bit as important as concurrence, its additive counterpart. (This distinction is that of multiplicative-additive linear logic or MALL, where *concurrence* and *orthocurrence* are called respectively *plus* and *tensor*, and a third operation *perp* is introduced whose action on processes is to interchange the perspectives of observer and observed, with the side effect of interchanging events and states.)

Our standard example of orthocurrence has long been that of trains passing through stations, which this paper spices up with examples involving pigeons choosing holes subject to various constraints. Further evidence for the utility of orthocurrence appears in Pratt (2000) and Rodriguez and Anger (2001), where Allen interval algebras for various models of time, previously obtained tediously by *ad hoc* manual methods (Anger and Rodriguez 1991; Rodriguez and Anger 1993a; Rodriguez and Anger 1993b), are all obtained uniformly and automatically using orthocurrence.

To these two assumptions of the standard model, one might add a third, discrete time. The standard model assumes that time passes in discrete steps, justified by the behaviour of digital systems, which are understood abstractly as so behaving. Yet flip-flops, Schmitt triggers, *etc.* are implemented from analogue transistors, signal processing and image processing involve the digitisation of analogue signals, and buses need to decide quickly which of two independently arriving pulses arrived first on two request lines in order to grant a shared resource to one requester without inadvertently also granting it to the other one. The transition-based view of computation has been coerced by many to this analogue viewpoint in recent years, but not gracefully, in the sense that the usual approach has been to interleave small intervals of the continuum, which only exacerbates the problems created by the assumptions of fixed granularity and sequential observation.

A suitable framework for this purpose that is more robust than interleaved intervals is provided by generalised metric spaces (Casley *et al.* 1991).

2. Existing models of behaviour

2.1. *Petri nets*

The earliest extant model of concurrency is the ***Petri net*** (Petri 1962), a structure consisting of statements[†] and transitions. Truth values of statements are natural numbers, and a statement holds when its truth is non-zero. A marking or *state* of a Petri net is a truth assignment to its statements. An *event* is the firing of a transition. Firing is regulated by associating to each transition two sets of statements, its preconditions and its postconditions. When all the preconditions of a transition hold it can fire, and when it does so, it decrements its preconditions and increments its postconditions.

A sequential run of a Petri net is a sequence of firings. A concurrent run is a so-called *occurrence* or *causal* net, an acyclic Petri net each of whose statements is postcondition to just one transition and precondition to just one other, whose transitions now constitute events in the sense that they can each fire at most once. Variations in width of a run express the varying degrees of concurrency during the run, that is, the number of transitions that can be firing concurrently.

2.2. *Mazurkiewicz traces*

In the same way that many different state automata can all exhibit the same sequential behaviour, so many different Petri nets can all manifest the same concurrent behaviour. One therefore seeks a more abstract notion of behaviour that minimises irrelevant implementation detail.

Just as formal languages abstract state automata by modelling the latter's runs as strings, so ***Mazurkiewicz traces*** (Mazurkiewicz 1977) abstract Petri nets by modelling their runs as equivalence classes of strings. Consider, for example, a Petri net consisting of two isolated transitions labelled **a** and **b**, respectively, with a transition being considered isolated when it has a true precondition and a false postcondition that are each shared with no other transition. Whereas the sequential behaviour of this Petri net would consist of the two strings **ab** and **ba**, its Mazurkiewicz behaviour would consist of the single Mazurkiewicz trace that results from identifying these two strings to indicate that a and b had fired independently.

The basis for this identification is *action independence*, defined as an irreflexive symmetric binary relation I on an alphabet Σ of actions. Action independence induces a congruence $\cong_I$ (with respect to concatenation) on the monoid Σ^* of all finite strings on Σ, namely the least congruence $\cong$ for which $\mathbf{ab} \cong \mathbf{ba}$ for all $(\mathbf{a}, \mathbf{b}) \in I$. (We distinguish between actions **a** as event *labels* and events a as action *instances* using boldface and italics, respectively.) That is, two strings are I-congruent when one can be obtained from the other by a

[†] *Stellen*, traditionally translated more literally as places.

series of exchanges of adjacent independent actions. The quotient $\Sigma^*/\cong_I$ consisting of the congruence classes of $\cong_I$ constitutes the monoid of all Mazurkiewicz traces over Σ induced by I. A concurrent behaviour is then a set of such traces, that is, a subset of $\Sigma^*/\cong_I$.

A fundamental limitation of Mazurkiewicz traces is that independence is global: if actions **a** and **b** are independent anywhere, they are independent everywhere. Consider the four actions one would use to model the transmission and receipt of bits through an asynchronous (buffered) channel, say $\mathbf{T_0}, \mathbf{T_1}$ (transmission of 0 or 1) with corresponding receipts $\mathbf{R_0}, \mathbf{R_1}$. Receipt of the first bit necessarily depends on its transmission, but for proper asynchrony that same receipt should be independent of the transmission of the second bit. So if both transmitted bits are the same, say zero, both bits are transmitted *via* the same action, namely $\mathbf{T_0}$. But then the action of receipt of the first bit, namely $\mathbf{R_0}$, must be both dependent on and independent of $\mathbf{T_0}$ as the common action of both the first and second transmitted bits, which is an impossibility for Mazurkiewicz traces.

2.3. *Pomsets*

Pomsets (Grabowski 1981; Pratt 1982; Pratt 1986; Gischer 1988) overcome this limitation of Mazurkiewicz traces by being based on independence not of actions but of action instances or *events*. A pomset or partially ordered multiset is a Σ-labelled poset $(A, \leqslant, \lambda)$ where $\leqslant$ partially orders a set A of events, with $a \leqslant b$ indicating that a necessarily happens before b, and $\lambda : A \to \Sigma$ labels each event $a \in A$ with an action $\lambda(a) \in \Sigma$. A string over Σ is the special case of a pomset that is finite and linearly ordered. A Mazurkiewicz trace in $\Sigma^*/\cong_I$ is the special case of a pomset that is finite and, for any pair a, b of distinct events, if a and b are order-incomparable, then $\lambda(a)$ and $\lambda(b)$ are independent, and if $\lambda(a)$ and $\lambda(b)$ are independent and $a \leqslant b$, there exists a third event c satisfying $a < c < b$. When I is empty all events are order-comparable, that is, the order is linear, in which case Mazurkiewicz traces are ordinary strings, as expected. Strings, Mazurkiewicz traces and pomsets thus form increasingly more general kinds of behaviours.

From the perspective of true concurrency, pomsets (and hence strings and Mazurkiewicz traces) represent behaviours that are not only deterministic but non-branching: there is no element of choice whatsoever. Whereas 'width' in an automaton results from disjunctive branching (just one branch is taken), the visually similar 'branches' in a pomset are all taken concurrently, thereby constituting conjunctive branching. We will shortly settle on processes containing both kinds of branching. In that setting it is both customary and natural to hide conjunctive 'branching' by lumping together strings, Mazurkiewicz traces and pomsets under the general heading of ***traces***, understood as deterministic non-branching behaviours, and to reserve the term 'branching' exclusively for its disjunctive form.

2.4. *Branching time and bisimulation*

Prior to the last two decades or so, non-determinism and choice were customarily modelled abstractly in terms of sets of traces denoting the possible alternative runs. The abstract behaviour of an automaton was taken to be the set of strings it accepted, and for a Petri net it was the set of its permitted firing sequences. This catered for non-determinism and branching by effectively making all choices blindly at the outset. To handle the case of a decision predicated on the outcome of a test performed during the behaviour, if that outcome turns out to conflict with the choice made at the beginning, the trace *blocks*, which means that it does not terminate successfully, but rather it is abandoned as having been a bad choice, like a prospector giving up on an unproductive mining claim. The distinction between successful termination and blocking is observable in the construct $\mathscr{A}\mathscr{B}$, perform $\mathscr{A}$ then $\mathscr{B}$, in that when process $\mathscr{A}$ terminates successfully process $\mathscr{B}$ may proceed, whereas if $\mathscr{A}$ blocks, so does $\mathscr{A}\mathscr{B}$, and $\mathscr{B}$ does not get to start.

Models of behaviour that make their decisions in the beginning in this way are said to obey ***trace semantics***. The litmus test for trace semantics is the equation $a(b+c) = ab+ac$, meaning that choosing one of b or c after performing a is the same as choosing one of ab (perform a then b) or ac. Students of automata theory will recognise this equation as holding for formal languages (sets of strings) where ab is concatenation and $a+b$ union, but it also holds for processes as sets of traces of any kind including pomsets. *Every model of computation considered thus far obeys trace semantics.*

Now it is common sense that a prematurely made decision may adversely limit our options by committing us to a path before sufficient information is available to justify that decision. In that respect $a(b+c)$ is more prudent than $ab+ac$, which makes $a(b+c) = ab+ac$ inappropriate.

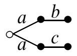
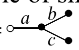
A finer equivalence than trace equivalence that recognises the impact of decision timing is *bisimilarity* (Milner 1980; Park 1981), which just satisfies $(a+b)c = ac+bc$, and not $a(b+c) = ab+ac$.

Bisimilarity is defined not on events but on states of an automaton or transition graph, *via* the notion of simulation. For the purpose of this definition we take an automaton to be a vertex-and-edge-labelled graph $\mathscr{X} = (X, E)$ whose vertices are states $x \in X$ labelled with predicates $\lambda_X(x)$ and whose edges in E are transitions from x to x' labelled with actions $\mathbf{a}$, denoted $x \xrightarrow{\mathbf{a}} x'$. A binary relation $R \subseteq X \times Y$ relating states of $\mathscr{X}$ to states of $\mathscr{Y}$ is a ***simulation*** between $\mathscr{X}$ and $\mathscr{Y}$ when for all (x, y) in R,

(i) $\lambda_X(x) = \lambda_Y(y)$, and

(ii) for every transition $x \xrightarrow{\mathbf{a}} x'$ in E_X there exists a like-labelled transition $y \xrightarrow{\mathbf{a}} y'$ in E_Y, the *simulating* transition, for which $(x', y') \in R$.

State y in $\mathscr{Y}$ ***simulates*** state x in $\mathscr{X}$ when there exists a simulation between $\mathscr{X}$ and $\mathscr{Y}$ containing (x, y). For automata with a specified initial state, when the initial state of $\mathscr{Y}$ simulates the initial state of $\mathscr{X}$, we say that $\mathscr{Y}$ simulates $\mathscr{X}$.

For an example of simulation consider the automaton $\mathscr{X} =$  realising the behaviour $ab+ac$ and $\mathscr{Y} =$  realising $a(b+c)$, with states labelled vacuously (the same predicate on every state). Each of their states is reached from its parent initial state by exactly one of the strings ϵ, a, ab, ac. Take $R \subseteq X \times Y$ to consist of all pairs (x, y) such that x and y are reached by the same string (one such pair for each state of $\mathscr{X}$, in fact R is a surjection from X onto Y that is injective except for identifying the two states of $\mathscr{X}$ reached by a). R is easily verified to be a simulation, and, furthermore, it relates initial states and so is a simulation of $\mathscr{X}$ by $\mathscr{Y}$. However, the converse of R is not a simulation of $\mathscr{Y}$ by $\mathscr{X}$ because the (x, y) in R for which x is the midpoint of ab is such that y has a c transition from it but x does not. Moreover, there exists no simulation at all of $\mathscr{Y}$ by $\mathscr{X}$, the price of $\mathscr{X}$'s imprudently early decision.

A ***bisimulation*** is a simulation whose converse is also a simulation. Processes with initial states are bisimilar when there is a bisimulation between them that makes their initial states bisimilar. Semantics making finer distinctions than trace equivalence have been provided for both CSP (Brookes *et al.* 1984) and CCS (Milner 1980). Such semantics are distinguished from trace or linear time semantics under the rubric of branching-time semantics, with bisimilarity being the finest such, at least for ordinary yes/no non-determinism as opposed, for example, to probabilistically governed choices.

Bisimilarity breaks the symmetry of time inherent not only in traditional automata theory based on sets of strings but also in accounts of concurrency based on sets of Mazurkiewicz traces and sets of pomsets – in general on sets of traces. It characterises branching time intensionally by making us write $(a + b)c \cong ac + bc$ expressing the bisimilarity of $(a + b)c$ and $ac + bc$ and $a(b + c) \ncong ab + ac$ for the non-bisimilarity of $a(b + c)$, and $ab + ac$ when an extensional characterisation would permit the conceptually simpler $(a + b)c = ac + bc$ and $a(b + c) \neq ab + ac$.

2.5. *Event structures*

Our program at this point is to characterise both concurrency and branching time extensionally. We take as our basic framework for this program that of event structures (Nielsen *et al.* 1981; Winskel 1980; Winskel 1986; Winskel 1988a).

An ***event structure***[†] $(A, \leqslant, \#)$ consists of a partial order $(A, \leqslant)$[‡] equipped with a symmetric irreflexive binary relation $\#$ of *conflict* satisfying the following condition.

Axiom ES. For all events $a, b, c \in A$, $a\#b$ and $b \leqslant c$ implies $a\#c$.

A ***configuration*** of an event structure $(A, \leqslant, \#)$ is a conflict-free downset $x \subset A$[§]. That is, given $a \in x$, if $b \leqslant a$, then $b \in x$ (the notion of downset), and if $a\#b$, then $b \notin x$ (the meaning of conflict).

The set X of all configurations of an event structure, called a *family* of configurations, can be understood as the states of an acyclic automaton realising the event structure. The initial state of this automaton is the empty set of states, which is vacuously conflict-free and a downset, and hence a configuration. Its transitions are the inclusions between configurations, and are oriented to pass from the smaller state to the larger.

Event structures as such are unlabelled. They may be labelled with labels drawn from an action alphabet Σ, exactly as with pomsets. A ***labelled event structure*** $(A, \leqslant, \#, \lambda)$ is an

[†] These are subsequently described as *prime coherent* event structures to distinguish this basic class from larger classes of event structures introduced later.

[‡] Note the absence of cardinality restrictions: A may be finite (even empty), countably infinite or uncountable.

[§] Configurations represent states. For uniformity, we shall write states as lower-case $x, y, z, \ldots$ independently of whether they take the form of vertices of a graph or configurations of an event structure. We reserve upper case X for sets of states.

event structure together with a labelling function $\lambda : A \to \Sigma$. As for pomsets, labelling specifies for each event the action it performs. Many events may perform the same action, for example, while a communication channel may transmit any number of bits, each transmission is an instance of one of two actions, $\mathbf{T_0}$ and $\mathbf{T_1}$, denoting transmission of 0 or 1, respectively.

Labels carry over to the family of configurations in a straightforward way. A transition from x to y is labelled with the multiset of labels of the events in $y - x$ (those events that happened during the passage from x to y). This multiset can be defined as the restriction of λ to $y - x$.

Labels make it easy to distinguish $\mathbf{a}||\mathbf{b}$ from $\mathbf{ab}+\mathbf{ab}$ and $\mathbf{a}(\mathbf{b}+\mathbf{c})$ from $\mathbf{ab}+\mathbf{ac}$. For the former we take $\mathbf{a}||\mathbf{b}$ to consist of two events labelled $\mathbf{a}$ and $\mathbf{b}$, respectively, and $\mathbf{ab}+\mathbf{ab}$ to consist of four events labelled $\mathbf{a}$, $\mathbf{b}$, $\mathbf{a}$ and $\mathbf{b}$, respectively, with the event order and with every event of one branch of the choice in conflict with every event of the other. For the latter we take $\mathbf{a}(\mathbf{b}+\mathbf{c})$ to consist of three events labelled $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$, with $\mathbf{a}$ preceding both $\mathbf{b}$ and $\mathbf{c}$, and with $\mathbf{b}$ and $\mathbf{c}$ in conflict, and we take $\mathbf{ab}+\mathbf{ac}$ to consist of the evident four-event event structure.

Now these distinctions are made by the questionable practice of breaking what is intuitively a single event into multiple events, the copies of which are then kept track of *via* the labels. The question then arises as to whether a more conservative accounting of events is possible without having to duplicate an event, which then needs to be kept track of with labels. We answer this in the affirmative by accounting for both concurrency and branching time entirely with unlabelled event structures. As will become clear in due course, Figure 1 shows that this is not possible with ordinary or dyadic event structures, but becomes possible when further qualia besides 0 and 1 are added to time, corresponding to the constructs *before* (may happen), *during* (is happening), *after* (has happened), and *instead of* (won't happen). The states themselves are those of *initialisation*, *transition*, *termination* and *cancellation*.

2.6. *Nature of prime coherent event structures*

Before embarking on this project, we will help to fix some ideas by biding a while with the dyadic case, and, in particular, with the original prime coherent subcase.

The nature of prime coherent event structures is brought out by the following properties, leading up to the observation of Corollary 5 that prime coherent event structures are extensional (distinct event structures have distinct interpretations).

Lemma 1. Every principal downset of an event structure is a configuration of that event structure.

(A downset is ***principal*** when it contains its sup, or, equivalently, when it contains an element a such that the downset can be expressed as $\{b|b \leqslant a\}$, written $\downarrow a$.)

Proof. If the statement were false, the downset would have to contain a conflicting pair both below a. Two applications of Axiom ES would then yield $a\#a$, which contradicts the irreflexivity of $\#$. $\square$

Corollary 2. $a \leqslant b$ and $a\#b$ are mutually exclusive.

Proof. If the statement were false, the principal downset $\downarrow b$ would not be a configuration. □

Lemma 3. If a and b do not appear together in any configuration of an event structure then $a\#b$ in that event structure.

Proof. If a and b do not appear together in any configuration, then $\downarrow a \cup \downarrow b$ must not be a configuration. But it is a downset, and hence can only fail to be a configuration by containing two events in conflict. By Lemma 1, one must be in $\downarrow a$ and the other in $\downarrow b$. Two applications of Axiom ES then yield $a\#b$. □

Lemma 4. If a does not appear in any configuration of an event structure without b, then $b \leqslant a$ in that event structure.

Proof. By Lemma 1, $\downarrow a$ is a configuration, which by hypothesis contains b, so $b \leqslant a$. □

Corollary 5. An event structure is uniquely determined by its family of configurations.

Proof. The proof is straightforward given Lemmas 3 and 4). □

Lemma 6. The number of event structures on a given set having one or two events is one or four, respectively.

Proof. A singleton admits only one partial order and one irreflexive binary relation. A doubleton admits three partial orders and two irreflexive symmetric binary relations for a total of six, but the two of these in which the events are in conflict and the order is linear are ruled out by Corollary 2. The four survivors are then seen by inspection to be event structures. □

The four two-event structures are succinctly characterised as those that omit either no configurations or one non-empty configuration. Apart from the two linearly ordered event structures, these are clearly non-isomorphic, so, up to isomorphism, there are only three event structures on two events.

2.7. *Event structures via logic*

Event structures have evolved with time and experience to successively larger classes, most notably through the work of Glynn Winskel (Winskel 1980; 1982; 1986; 1988b). A uniform framework from which to view this evolution casts an event structure as a Boolean operation[†] whose satisfying assignments meet certain plausible restrictions. One such restriction might insist on the all-zero assignment (the initial state) being among the satisfying assignments. Another might require that every satisfying assignment be reachable from the all-zero assignment *via* a *monotone Hamming* path through the

[†] Boolean operations are used in preference to Boolean formulas because the former as restricted are in one–one correspondence with event structures, while for the latter the correspondence is many–one.

satisfying assignments, that is, one that proceeds by changing one atom at a time from false to true. (Prime coherent event structures meet both of these restrictions.) The evolution of more general event structures can then be described simply as the gradual removal of such restrictions.

In more detail, an event structure $\mathscr{E}$ on a set A of events is viewed as a Boolean operation $f_{\mathscr{E}}$ on A, namely a function of type $2^A \to 2$, or, equivalently, a set of sets of type 2^{2^A}. The elements of A are reinterpreted as atoms or propositional variables. Event a is understood as an atomic proposition P_a expressing 'a has happened'. We abbreviate P_a to a and rely on context to disambiguate. The configurations of $\mathscr{E}$ then become the *satisfying assignments* of $f_{\mathscr{E}}$, that is, those assignments $\alpha : A \to 2$ of truth values to atoms such that $f_{\mathscr{E}}(\alpha) = 1$.

Prime coherent event structures, the class treated above, are translated as follows. Each temporal precedence $a \leqslant b$ is rendered as the implication $b \to a$ between atoms (or $\overline{a} \to \overline{b}$ between literals), namely, if b has happened, then a has happened. (The implication arrow thus points backwards in time.) Each conflict $a\#b$ is expressed as the proposition $\overline{a \wedge b}$, or $a \to \overline{b}$, or $b \to \overline{a}$. And Axiom ES simply disappears, being absorbed into the general Boolean framework as the application of transitivity to $a \to \overline{b}$ and $\overline{b} \to \overline{c}$.

We then define $T_{\mathscr{E}}$ to be the least literal-implication theory on A containing the above implications of $\mathscr{E}$. To this end, we define a ***literal-implication theory on*** A to be a reflexively transitively closed set of implications $P \to Q$ between literals P, Q that are each of the form a or $\overline{a}$ for some $a \in A$. We further require that such a theory also be closed under *modus tollens*, meaning that if $P \to Q$ is in the theory, then so is $\overline{Q} \to \overline{P}$.

The associated Boolean operation $f_{\mathscr{E}}$ holds (evaluates to 1) just where every implication in $T_{\mathscr{E}}$ holds, that is, $f_{\mathscr{E}}$ is the denotation of $\bigwedge T_{\mathscr{E}}$.

We say that a Boolean formula on A holds in an event structure $\mathscr{E}$ when it is a consequence of the Boolean operation $f_{\mathscr{E}}$ associated to $\mathscr{E}$, that is, when it holds in every assignment (of truth values to atoms of A) satisfying $f_{\mathscr{E}}$.

Any literal-implication theory of a prime coherent event structure must satisfy the following three conditions. It may not contain any implication $\overline{a} \to b$ from a negative literal to a positive (equivalently disjunctions of positive literals), because these are falsified by the empty (initial) configuration. It may not contain any implication $a \to \overline{a}$, as this expresses $a\#a$, which, by Lemma 3, violates irreflexivity of $\#$. And it may not contain both implications $a \to b$ and $b \to a$ (equivalently $a \equiv b$ cannot hold in $\mathscr{E}$), because of Lemma 4 and antisymmetry of $\leqslant$.

Theorem 7. Every literal implication theory satisfying the above three conditions arises as the translation of some prime coherent event structure.

Proof. Given such a theory T on A, construct the event structure $\mathscr{E}$ on A for which $a \leqslant b$ holds when $b \to a$ is in T, and for which $a\#b$ holds when $a \to \overline{b}$ is in T. We claim that $T = T_{\mathscr{E}}$. This follows because for every $a \in A$, we have $a \to a$ and $\overline{a} \to \overline{a}$ must be present by reflexivity, while $a \to \overline{a}$ and $\overline{a} \to a$ must be absent by two of the conditions. For all $a \neq b$ we need only consider all implications from signed a to signed b, and no implications from signed b to signed a, since the latter set is completely determined from the former by *modus tollens*. $\overline{a} \to b$ is forbidden, and the conditions allow at most one of

$a \to b$, $\overline{a} \to \overline{b}$ (equivalent to $b \to a$), and $a \to \overline{b}$. That one will clearly be in $T_{\mathscr{E}}$ if and only if it was in T. □

A useful formula not expressible with prime coherent event structures is $b \to (a \vee c)$. This asserts that b cannot happen until one of a or c has happened, for example, the vending machine will not produce a soft drink until either a dollar bill or four quarters are inserted. Yet another is $(a \wedge c) \to b$, which asserts that a and c are in conflict (cannot both happen) until b happens, for example, you cannot buy both the candy and the pen you want with the dollar you have until you get a second dollar. This could be called either temporary conflict or deferred concurrency. Neither of these are expressible as a conjunction of literal implications. In view of the evident utility of such constructs, it is natural to look for ways to broaden the definition of event structures to admit them.

In this view of the evolution of the class of event structures by removal of the less natural or undesired restrictions, there is no clear-cut boundary for what counts as 'natural'. This makes the inevitable limit that of no restrictions, that is, any infinitary Boolean operation may be allowed. This is the point of view taken in Gupta and Pratt (1993) and van Glabbeek and Plotkin (1995), which study event structures at this level of generality from the viewpoint of extensional Chu spaces over a two-letter alphabet $K = \{0, 1\}$, a connection that is not the central point of this paper but whose notation and concepts we shall borrow from.

Taking this viewpoint here, we may summarise the development thus far by characterising event structures on a set A of events as precisely the Boolean operations on A, that is, entities of type 2^{2^A}. The automaton associated with such an event structure has for its states the set X of satisfying assignments of that operation, and its multi-event transitions, or ***steps***, are all pairs (x, y) of states for which $x \leqslant y$ coordinatewise: each event may stay in its present state, or pass from 0 to 1, but not from 1 to 0. Note that the step relation is transitive for now, but this will change later when we introduce $\lrcorner\!\ulcorner$.

We make a point of distinguishing $f : 2^A \to 2$ from $g : 2^B \to 2$ for $A \neq B$ even when f and g are both constantly true or both constantly false. Now $2^A \to 2$ and 2^{2^A} are isomorphic *via* the obvious pairing of $f : 2^A \to 2$ with $(A, \{x \in 2^A \mid f(x) = 1\})$. We shall abuse type and speak as though f and its matching (A, X) were identical. From now on we treat event structures and (abstract) Boolean propositions as one and the same.

One benefit of this identification is simplification of concepts. One drawback that we shall pursue in more detail shortly is that the naming conventions of process algebra and logic are not fully compatible: the constant 0 and the atom a denote different entities in each. We deal with this by assuming the process algebra interpretation by default, and indicate any exceptions explicitly.

One way of presenting (A, X) is as an $A \times X$ matrix whose entry at (a, x) is the value of a in state x. Thus the independent behaviour $a||b$ of two states a, b is presented as the matrix $\begin{array}{c|c|} a & 0101 \\ b & 0011 \end{array}$, while the sequence ab removes the 01 state to give $\begin{array}{c|c|} a & 011 \\ b & 001 \end{array}$. (This matrix viewpoint brings out the duality of events and states mentioned in the introduction better than either the event structure or Boolean formula viewpoint.) Since the columns arise as subsets of A, any two columns of such a matrix must be distinct (so there can be at most exponentially many more columns than rows); this, however, is the only restriction on such matrices for the case of general Boolean operations.

We call two formulas $\mathscr{A}, \mathscr{B}$ ***isomorphic*** when there exists a bijection $g : A \to B$, understood as a 1–1 renaming of atoms, such that $X = \{y \circ g \mid y \in Y\}$, that is, when they have the same satisfying assignments modulo the renaming.

We conclude this section with a systematic enumeration of the types this paper is primarily concerned with, namely, processes, events, states and values.

At the top level are *processes* $\mathscr{A}, \mathscr{B}, \mathscr{C}$. In a more categorically oriented version of this paper these would form a category furnished with structure *via* morphisms between processes. In this version, however, processes are organised into a process algebra *via* the operations treated in the next section.

The principal constituents of a process are its *events* a, b, c, forming a set A, and its *states* x, y, z forming a set X. Thus far we have viewed states extensionally as sets of events, making A the only primitive set with $X \subseteq 2^A$. However, they can also be viewed intensionally as independent entities sibling to events, so that we now have two primitive sets A and X. In this case we no longer have set membership defining the relationship between events and states; instead we provide its intensional counterpart explicitly, as an arbitrary binary relation or matrix $R \subseteq A \times X$ between A and X. Natural notations for $R(a, x)$ are $a.x$, in the way that mathematicians write inner product in a Hilbert space, or $\langle a \mid x \rangle$, as physicists write it, with the value, however, being a truth value rather than a complex number.

The extensional view is the special case of the intensional view with $X \subseteq 2^A$ and $R = \in$. In this paper we forego the appealing symmetry of the intensional view[†] and stick to the extensional view, whose benefit is that a process is just a pair (A, X) with $X \subseteq 2^A$ instead of a triple (A, R, X) with $R \subseteq A \times X$.

Implicit in the above are the values 0 and 1 taken by the formulas $a \in x$ and $R(a, x)$. As these values will later be joined by $\lrcorner$ and $\times$, we make them more explicit here, collecting them as the set $K = \{0, 1\}$ of *values*. We omit the customary prefix “truth” as these values are schizophrenic, having temporal significance when varying down a column of R and propositional or truth significance when varying across a row.

Summarising in reverse order, the types are *values*, *events*, *states*, *processes* and *process operations*.

Besides the above, we have also encountered *labels* $\mathbf{a}, \mathbf{b}, \mathbf{c}$ forming an alphabet Σ. However, these only appear with labelled event structures, which we touch on only tangentially in this paper. We also have *logical operations*, which play an important role in expressiveness, permitting every process to be named even when $|K| = 4$, as we shall show later, but which are less natural for programming purposes than the process operations, which are the topic of the next section.

2.8. *Process algebra*

Process algebra (Bergstra and Klop 1984; Baeten and Weijland 1990; Bergstra *et al.* 2000) provides a way of assembling complex processes from simpler ones. This is in contrast to

[†] The extensional view can be symmetrised by forbidding repeated rows (‘little chu’ in the jargon of Chu spaces (Barr 1991)), thus allowing events to be viewed as sets of states.

Amir Pnueli's temporal logic (Pnueli 1977), which describes a single complex process from the perspective of a single neutral observer of that process's universe. This distinction has been described by Pnueli as exogenous and endogenous specification, respectively, the latter constituting the crucial element of temporal logic that distinguishes it from dynamic logic (Pratt 1976; Harel *et al.* 2000), which is a similar modal logic of programs that extends temporal logic with the ability to state properties of compositional or algebraic programs.

Temporal logic's neutral-observer viewpoint intrinsically entails an interleaving view of the progress of the universe. Process algebra restores compositionality of programs in a way that works equally well for interleaving semantics and true concurrency, even when both the observer and observed are distributed, as catered for by orthocurrence. Process algebra is a popular subject with a rich literature, many workshops, a comprehensive handbook (Bergstra *et al.* 2000), and many skeptics whose purposes are adequately served by the interleaving viewpoint of concurrency as discussed in Section 1.3.

Before considering the operations for growing bigger processes from smaller, let us consider the smallest processes, that is, those without a plurality of events, of which there are six. The two without any events are (in process algebra notation) 0 and $\bullet$, having one and no states, respectively, corresponding to the truth constants (zeroary operations) 1 and 0, respectively. (So without any events at all, we already have a naming discrepancy between the notations of process algebra and logic.)

The four processes with one event are, in logical notation, a, $\overline{a}$, 1 and 0 (the last two as constant unary operations). Formula a holds only when event a is 1, and therefore denotes the 'stuck-at-one' process, which starts in the state in which a has already happened and stays there. Formula $\overline{a}$ is the counterpart for the process that stays in one state in which a has not happened. Formula 1 is the unconstrained process, having both possible states 0 and 1. Formula 0 is the inconsistent process, having no states.

Of the above four, process algebra notation only offers a name for the third, the unconstrained process. When processes are named only up to isomorphism, the name 1 has the same denotation in process algebra and logic[†]. In this paper, however, we will generally not consider processes as being defined only up to isomorphism. Instead we name one-event processes by their one event, since when there are several events a, b, c floating around, we will want to keep track of which processes have which event. We therefore call each such process by the name of its one event.

We then have a second naming discrepancy: logically a denotes the already-done process, whereas process algebraically a denotes the unconstrained process.

We now define four basic process algebra connectives: *concurrence* $\mathscr{A}||\mathscr{B}$, *sequence* $\mathscr{A}\mathscr{B}$, *choice* $\mathscr{A}+\mathscr{B}$ and *orthocurrence* $\mathscr{A}\otimes\mathscr{B}$. For all four definitions we assume $\mathscr{A}=(A,X)$ and $\mathscr{B}=(B,Y)$ where $X\subseteq 2^A$ and $Y\subseteq 2^B$. Given our identification of event structures with Boolean operations, this makes process algebra connectives be logical connectives also, though not necessarily *via* the most obvious connections – $\mathscr{A}+\mathscr{B}$ is not $\mathscr{A}\vee\mathscr{B}$, for

[†] Well, almost, since the latter requires the additional information that this is the unary constant 1 as opposed to some other arity. This process, incidentally, is the tensor unit when these processes are organised into a $*$-autonomous category (Barr 1979; Gupta and Pratt 1993) to form a model of linear logic (Girard 1987).

example, concurrent process algebra does not enjoy the logical simplicity of sequential dynamic logic (Pratt 1976).

We define $\mathscr{A} \wedge \mathscr{B}$ as $(A \cup B, \{z \in 2^{A \cup B} \mid z \restriction A \in X \;\wedge\; z \restriction B \in Y\})$, where $z \restriction A$ denotes the restriction of z (whose domain is $A \cup B$) to the subdomain A. One extreme of this is when A and B are disjoint, in which case $\mathscr{A} \wedge \mathscr{B}$ is isomorphic to $(A + B, X \times Y)$ (understanding $(x, y)(a) = x(a)$ and $(x, y)(b) = y(b)$), which is coproduct in the category of extensional Chu spaces over 2. The other extreme is when $A = B$, for which $\mathscr{A} \wedge \mathscr{B}$ simplifies to $(A, X \cap Y)$, that is, ordinary conjunction as intersection of state sets. Hence conjunction is idempotent: $\mathscr{A} \wedge \mathscr{A} = \mathscr{A}$. Similarly, we define $\mathscr{A} \vee \mathscr{B}$ as $(A \cup B, \{z \in 2^{A \cup B} \mid z \restriction A \in X \;\vee\; z \restriction B \in Y\})$, which is similarly idempotent. Both $\wedge$ and $\vee$ are clearly also commutative, and almost as clearly associative.

For occasional use in this paper we also define 'external' implication $\mathscr{A} \vdash \mathscr{B}$, $\mathscr{A}$ *entails* $\mathscr{B}$, as simply $X \subseteq Y$. This only makes sense when $A = B$ but we shall only need it for this case in what follows. (There is more than one internal implication, one being $\mathscr{A} \multimap \mathscr{B}$, definable in terms of orthocurrence and perp as $(\mathscr{A} \otimes \mathscr{B}^{\perp})^{\perp}$, but it is not needed for this paper.)

Concurrence $\mathscr{A} || \mathscr{B}$ is defined as $\mathscr{A} \wedge \mathscr{B}$. This makes $\mathscr{A} || \mathscr{B}$ the joint behaviour of $\mathscr{A}$ and $\mathscr{B}$: the events of both are performed subject to the constraints of each. Concurrence inherits the commutativity, associativity and idempotence of $\wedge$; in particular, $a || a = a$, a corollary of this being an algebra of unlabelled events. We shall discuss this disconcerting equation after treating the other operations.

In the case $\mathscr{A} = a$, $\mathscr{B} = b$ (viewing a and b as processes, not formulas, which is the practice we follow for process algebra) there are no constraints on a or b separately, and hence $a || b$ is equally unconstrained, that is, is the constantly true binary operation.

Sequence $\mathscr{A}\mathscr{B}$ is defined as $\mathscr{A} \wedge \mathscr{B} \wedge (B = 0 \vee \checkmark\mathscr{A})$. Here $B = 0$ denotes the process $(B, \{0^B\})$ having just the one all-zero state, which is expressible as $\bigvee_i \overline{b_i}$ taken over all atoms $b_i \in B$. However, $\checkmark\mathscr{A}$ is a more delicate notion, which we define for this dyadic case as $(A, \{x \in X \mid \forall z \in X[x \leqslant z \to x = z]\})$. So $\checkmark\mathscr{A}$ holds at just those states at which $\mathscr{A}$ holds and for which there is no greater (later) state at which $\mathscr{A}$ holds; these are considered the final states of $\mathscr{A}$. The equation $B = 0 \vee \checkmark\mathscr{A}$ holds at those states where either $\mathscr{B}$ has not yet started or $\mathscr{A}$ has finished. The meaning of $\mathscr{A}\mathscr{B}$ is therefore $\mathscr{A} || \mathscr{B}$ subject to the additional constraint that $\mathscr{B}$ remain in its initial state until $\mathscr{A}$ has entered a final state.

Whereas $B = 0$ is a local or first-order notion of initiality, $\checkmark\mathscr{A}$ is a global or second-order notion of finality necessitated by the limited scope of dyadic logic, $K = \{0, 1\}$. The *cancel* state $\times$ introduced later will permit a simpler and more robust first-order notion of termination.

When $\mathscr{A} = a$ and $\mathscr{B} = b$ the corresponding formula simplifies to $1 \wedge 1 \wedge (\overline{b} \vee a)$ (1 denoting *true* when used in formulas), or $b \to a$. This suggests the logical interpretation of sequence $\mathscr{A}\mathscr{B}$ as a new internal implication $\mathscr{B} \to \mathscr{A}$.

Just as $a || a = a$, we also have $aa = a$, unlike the situation with the string **aa**, which in the present context is understood as two consecutive actions constituting a labelled event structure with two distinct events. Viewed in this light, the notational conventions of automata theory make the letters of a regular expression such as $(\mathbf{a}(\mathbf{a} + \mathbf{b}))^*$ actions rather than events; a finite regular expression understood as a labelled event structure will always have finitely many actions, but, in general, infinitely many action instances or events.

Unlike concurrence, the above definition of sequence makes it not idempotent but only transitive ($\mathscr{A}\mathscr{A} \vdash \mathscr{A}$), witness $\mathscr{A} = ab$, which allows $a = 1$, $b = 0$, unlike $abab$, which forces $a = b$. It can be made idempotent by replacing $B = 0$ by $B - A = 0$ in the definition, but this destroys associativity: $a(ba)$ is ab while $a(ba)$ is $a \equiv b$. We will defer the further pursuit of this issue until we add $\times$ to K, which permits a better definition of $\checkmark\mathscr{A}$. Sequence is, of course, not commutative.

Normally, concurrence and sequence are defined in terms of the disjoint union or marked sum of their event sets. This is the appropriate combination in the case of labelled event structures where the labels are the primary identifying features and the identities of the individual events fade into the background. The present approach is therefore something of a novelty, especially the resulting idempotence or near-idempotence of these operators, which are customarily thought of as analogous more to addition than to union. The difference is in our focus on events a, b instead of actions $\mathbf{a}, \mathbf{b}$. Whereas multiple references to an event a all denote the same event, multiple references to an action $\mathbf{a}$ are understood to denote distinct instances of that action, that is, distinct events. Thus, whereas aa denotes the single event a preceding itself (which is vacuous by reflexivity of precedence), $\mathbf{aa}$ denotes two instances of action $\mathbf{a}$, with one preceding the other. This paper restricts its attention to events.

Choice $\mathscr{A} + \mathscr{B}$ is defined as

$$A \cup B = 0 \;\vee\; (\mathscr{A} \neq 0 \;\wedge\; B - A = 0) \;\vee\; (\mathscr{B} \neq 0 \;\wedge\; A - B = 0).$$

Here $A = 0$ denotes $(A, \{0^A\})$ (the process having just the one state 0^A in which all events are zero), while $\mathscr{A} \neq 0$ denotes $(A, X - \{0^A\})$ ($\mathscr{A}$ stripped of its zero state if any – '$\mathscr{A}$ under way'). The states of $\mathscr{A} + \mathscr{B}$ can therefore be partitioned into three disjoint blocks:

(1) the initial state in which all events of $A \cup B$ are zero;

(2) those states whose restriction to A is a non-zero state of $\mathscr{A}$ and for which those events of B not in A are all zero;

(3) states as in (2), but with $\mathscr{A}$ and $\mathscr{B}$ interchanged.

The role of the initial state is to permit $\mathscr{A} + \mathscr{B}$ to remain undecided prior to taking its turn. Once it is $\mathscr{A} + \mathscr{B}$'s turn, one of $\mathscr{A}$ or $\mathscr{B}$ must be selected and all events absent from the selected process cannot occur.

Substituting a for $\mathscr{A}$ and b for $\mathscr{B}$ in this definition of choice yields $(\overline{a} \wedge \overline{b}) \vee (a \wedge \overline{b}) \vee (b \wedge \overline{a})$, that is, $\neg(a \wedge b)$, expressing conflict $a \# b$.

For A, B disjoint and at least one of $\mathscr{A}$ or $\mathscr{B}$ having the all-zero state, $\mathscr{A} + \mathscr{B}$ coincides with the choice operation $\mathscr{A} \sqcup \mathscr{B}$ defined in Gupta and Pratt (1993) and Gupta (1994), namely, as a 2×2 block matrix whose two diagonal blocks are $\mathscr{A}$ and $\mathscr{B}$ and whose off-diagonal blocks are all zero.

At the other extreme, $A = B$, we have $\mathscr{A} + \mathscr{A} = \mathscr{A} + 0$ (where 0 is the process with no events and one state), and if $\mathscr{A}$ has the zero state we have $\mathscr{A} + \mathscr{A} = \mathscr{A}$. So choice is idempotent up to presence of the zero state. By the symmetry of the definition, it is commutative. To see that it is associative, we may describe $\mathscr{A}+\mathscr{B}+\mathscr{C}$ as having for its non-zero states the non-zero states of exactly one of the three arguments, with all events not in that argument held at zero.

Orthocurrence $\mathscr{A}\otimes\mathscr{B}$ is $(A\times B, \{z\in K^{A\times B} \mid z(\lambda,\forall)\in X \wedge z(\forall,\lambda)\in Y\})$, where $z(\lambda,\forall)$ denotes $\lambda a.z(a,b)$ with b universally quantified in the containing proposition, and, dually, for $z(\forall,\lambda)$. This makes $z(\lambda,\forall)$ and $z(\forall,\lambda)$ the columns and rows, respectively, of the $A\times B$ matrix z, making z, in effect, a 'bilinear' form on $A\times B$, which is analogous to Halmos' approach to defining tensor product of vector spaces (Halmos 1974, Section 25). The states of $\mathscr{A}\otimes\mathscr{B}$ may be thought of as all $A\times B$ crosswords that can be formed by taking the states of $\mathscr{A}$ as the 'down' dictionary and the states of $\mathscr{B}$ as the 'across' dictionary.

For $\mathscr{A}=a$, $\mathscr{B}=b$, orthocurrence gives an unconstrained process consisting of the single event (a,b). Any unconstrained one-event process acts as the unit for orthocurrence, and as such corresponds to the linear logic constant 1.

Orthocurrence is a versatile operation, which extends smoothly to ternary ($\mathscr{A}\otimes\mathscr{B}\otimes\mathscr{C}$) and higher arities as well to continuous (real-valued) time (Casley *et al.* 1991). It is associative because the states of $\mathscr{A}\otimes\mathscr{B}\otimes\mathscr{C}$, viewed as 'bricks', are constrained symmetrically by each of X, Y and Z along the respective axis. It is symmetric in the sense of being commutative up to isomorphism. It is not at all idempotent, however.

Orthocurrence can be understood as either interaction $\mathscr{A}\otimes\mathscr{B}$ or observation $\mathscr{A}\multimap\mathscr{B}=(\mathscr{A}\otimes\mathscr{B}^{\perp})^{\perp}$, where $\mathscr{B}^{\perp}$ is the transpose or dual of $\mathscr{B}$ and $\multimap$ is the previously mentioned internal implication (Pratt 2001). As interaction, it describes the collision of systems of particles or events, while as observation it describes the dependence of view on viewpoint: the state $z(a,\lambda)$ of $\mathscr{B}$ in $\mathscr{A}\otimes\mathscr{B}$ (or point or event $z(a,\lambda)$ of $\mathscr{B}$ in $\mathscr{A}\multimap\mathscr{B}$) as seen from $\mathscr{A}$ depends on which point a of A it is seen from. Whereas interaction is an explicitly symmetric notion, observation is an ostensibly asymmetric notion with an implicit symmetry corollary to the linear logic isomorphism $\mathscr{A}\multimap\mathscr{B}\cong\mathscr{B}^{\perp}\multimap\mathscr{A}^{\perp}$. Although orthocurrence is not customarily regarded as one of the basic process algebra operations, the author has maintained for many years (Pratt 1985; Pratt 1986; Casley *et al.* 1991) that it should be, long before realising (Gupta and Pratt 1993) that this created a compelling link between process algebra and linear logic.

The operations of concurrence and orthocurrence make no assumptions about K other than that it is a set. However, orthocurrence differs from concurrence (as defined in this paper) in that it makes no structural difference whether the two arguments are disjoint. Sequence depends on state 0 being viewed as initial and state 1 as final, and also involves a second-order notion of finality for process states. Choice depends only on state 0 being initial.

The definitions are for the unlabelled case. We have paid little attention to the labelled case because labelling is a simple process with no impact at all on the portion of process algebra treated here. Labels for the events of processes built with concurrence, sequence and choice are straightforward because events retain their identity between the inputs and output of those operations: the labels on the events of the result are simply those on the events of the inputs. For orthocurrence, labelling is defined as $\lambda(a,b)=(\lambda(a),\lambda(b))$, namely the label at a pair of events is the pair consisting of their respective labels as provided in the processes input to orthocurrence.

Table 1. *Two-valued process algebra.*

a	$\begin{array}{l l} a & 01 \end{array}$	$ba + ca$	$\begin{array}{l l} a & 00101 \\ b & 01100 \\ c & 00011 \end{array}$
$a \Vert b$	$\begin{array}{l l} a & 0101 \\ b & 0011 \end{array}$	$a(b + c)$	$\begin{array}{l l} a & 0111 \\ b & 0010 \\ c & 0001 \end{array}$
ab	$\begin{array}{l l} a & 011 \\ b & 001 \end{array}$	$ab + ac$	$\begin{array}{l l} a & 0111 \\ b & 0010 \\ c & 0001 \end{array}$
$a + b$	$\begin{array}{l l} a & 010 \\ b & 001 \end{array}$	$ab + ba$	$\begin{array}{l l} a & 0101 \\ b & 0011 \end{array}$
$a \Vert (b + c)$	$\begin{array}{l l} a & 010101 \\ b & 001100 \\ c & 000011 \end{array}$	$ab \otimes cd$	$\begin{array}{l l} ac & 011111 \\ ad & 000111 \\ bc & 001011 \\ bd & 000001 \end{array}$
$(b + c)a$	$\begin{array}{l l} a & 00101 \\ b & 01100 \\ c & 00011 \end{array}$	$(a + b) \otimes (c + d)$	$\begin{array}{l l} ac & 0000101 \\ ad & 0001010 \\ bc & 0010010 \\ bd & 0100001 \end{array}$

We have not suggested any particular choice of iterative or recursive construct because the topic even of least fixed points (let alone greatest, optimal, *etc.*) of terms built up from processes understood as arbitrary Boolean operations using the above process algebra connectives is sufficiently substantial as to warrant its own paper. Here the reader will need to settle for the account of $\mathbf{a}^*$, $(\mathbf{ab})^*$, and $\mathbf{a} + \mathbf{b}^*$ in Section 3.4 to get at least some idea of what an infinitely generated process should look like, and how it is affected by the additional states $\lrcorner$ and $\times$.

The original Chu semantics took the point of view when combining processes that they had disjoint sets of events, or, more precisely, that events from distinct processes were unique entities bearing no relationship to each other. The recognition here that processes may share events permits a straightforward account of terms such as $ba + ca$ and $ab + ac$. Such sharing is certainly expressible categorically with pushouts in place of coproducts, but ordinary union and intersection achieve the same end with less fuss and greater familiarity to most.

These definitions give rise to the semantics shown in Table 1 below for particular formulas in atoms a, b, c – the semantics of each formula is expressed as a matrix whose rows are the atoms and whose columns are the permitted states (the same notation as used for Chu spaces).

Note that this process algebra is an algebra of *unlabelled* event structures, in the sense that its variables range not over actions **a** but events a. Thus, in expressions such as $ab + ac$ and $ab + ba$ containing more than one occurrence of a variable, all instances of that variable refer not just to the same action but to the same event.

The atomic process a has two states: 0 for not done and 1 for done. The states of $a||b$ are all subsets of $\{a, b\}$. ab forbids the state 01 in which b happened without a, while $a + b$ forbids the state 11 in which both a and b happened. The set of states of $a||(b + c)$ is (isomorphic to) the cartesian product of the state sets of a and $b + c$, giving six states. $(b + c)a$ forbids the state 100 of $a||(b + c)$ in which a has happened and neither of b or c have happened; this is equivalent to $ba + ca$, and not at all controversial. $a(b + c)$ similarly forbids 010 and 001, and similarly is equivalent to $ab + ac$, which is consistent with trace semantics but not with bisimilarity. And $ab + ba$ is equivalent to $a||b$, which is consistent with interleaving concurrency but not with true concurrency.

The orthocurrence $ab \otimes cd$ describes the flow of ab through cd. Each of a and b passes each of c and d, and these four visits constitute the events of $ab \otimes cd$, which are (a, c), (a, d), (b, c) and (b, d), respectively. If a and b are trains and c and d stations, this describes two trains moving along a track between two stations. As can be seen from the states, train a arrives at station c first, then a arrives at d concurrently with b visiting c, and finally b arrives at d.

The orthocurrence $(a + b) \otimes (c + d)$ realises the analogous process with choice in place of sequence. If a and b are pigeons and c and d are pigeonholes, then $a + b$ represents choosing a pigeon (chosen at each hole) while $c + d$ represents the choice of hole (chosen by each pigeon). With these constraints we can either put pigeon a in hole c and pigeon b in hole d, or a in d and b in c, as reflected in the two final states.

A variant of the one-pigeon-one-hole discipline allows multiple pigeons per hole, while not permitting the same pigeon in two holes simultaneously. With two pigeons a, b and two holes c, d, the only way this can happen is as $(a||b) \otimes (c + d)$, which works out to be

$$\begin{array}{r|c|}\hline ac & 000010011 \\ ad & 000101100 \\ bc & 001000101 \\ bd & 010001010 \\ \hline\end{array}.$$

The final states are the last four. Pigeon a has to be in either hole c or hole d, and likewise for pigeon b, giving a total of four possibilities.

We have tacitly capitalised on two-valued logic here. While the final states of $(a||b) \otimes (c + d)$ are as claimed, they are constructed from states of $a||b$ that are not always final (but $c + d$ is kosher). The problem arises in any hole that does not receive two pigeons. The one final state of $a||b$ has both pigeons, which is fine for a hole receiving two pigeons but not one receiving one or no pigeons. We could try to fix this by replacing $a||b$ by $0 + a + b + (a||b)$ so as to allow each hole to receive anywhere from 0 to two pigeons, but when we do so, we find that nothing changes. $K = \{0, 1\}$ is too small to appreciate these fine distinctions, allowing us to be sloppy without penalty. The introduction of $\times$ later will oblige us to count pigeons more carefully.

Three pigeons a, b, c any two of which can occupy the same hole can be described as

$$(a||b) + (b||c) + (c||a) = \begin{array}{r|c|}\hline a & 0101010 \\ b & 0011001 \\ c & 0000111 \\ \hline\end{array}.$$

But we still cannot put a pigeon in two holes d, e at the same time, so we have

$$((a||b) + (b||c) + (c||a)) \otimes (d + e) = \begin{array}{r|c|}\hline ad & 0000001000000001111000111 \\ ae & 0000010000011110000111000 \\ bd & 0000100001100010001011001 \\ be & 0001000110001000100100110 \\ cd & 0010000010100100010101010 \\ ce & 0100000101010001000010101 \\ \hline\end{array}.$$

This process has six final states (the six at the right, which each have three 1's), which correspond to the three choices of a lone pigeon and the two holes it can go in.

Again we have the sloppiness permitted by $K = \{0, 1\}$ that $(a||b) + (b||c) + (c||a)$ is indistinguishable from $a + b + c + (a||b) + (b||c) + (c||a)$. Again the appearance of $\times$ later will oblige greater care in counting pigeons.

2.9. *Higher dimensional automata*

An ordinary state automaton or transition system can be viewed as an edge-labelled graph. Geometrically, the vertices and edges are both cells, of dimension 0 and 1, respectively, with the vertices supplying the edges with boundaries, with two per edge except in the case of self-loops.

A ***higher dimensional automaton*** (Pratt 1991) or HDA is an automaton that admits cells of higher dimension, which have as their boundaries cells of lower dimension representing the immediately preceding or following more quiescent states. Related prior art includes Papadimitriou's geometric treatment of concurrency control (Papadimitriou 1986), van Glabbeek and Vaandrager's ST-bisimulation (van Glabbeek and Vaandrager 1987), and the deterministic asynchronous automata of Shields (Shields 1985). Since their introduction, higher dimensional automata have been treated by a number of authors (van Glabbeek 1991; Goubault and Jensen 1992; Goubault and Cridlig 1993; Goubault 1993; Gunawardena 1994; Goubault 1995a; Goubault 1995b; Goubault 1996a; Goubault 1996b; Sassone and Cattani 1996; Buckland and Johnson 1996; Takayama 1996; Sassone and Cattani 1996; Fajstrup *et al.* 1998; Pratt 2000), and have led to two conferences and a special issue of this journal (Goubault 2000).

The basic example of an HDA is the representation of independence $a||b$ as a solid square. The boundary of this square represents the sequential or interleaving interpretation of $a||b$, which is equivalent to $ab + ba$, while the interior represents the temporally overlapping or independent or concurrent occurrences of a and b.

More generally, a cubic cell of dimension n expresses the simultaneous occurrence of n events, and its boundaries, as cubes of dimension less than n, represent simultaneous occurrence of some of those events with the remaining events, which have each either not yet started or are completed.

In full generality, a higher dimensional automaton is a *cubical complex*, namely a set of cubes permitted to share boundaries, which create the possibilities of sequencing, branching, rejoining and cycles[†]. The case of all cells of dimension at most one is just the case of ordinary graphs (with multiple edges permitted between any two vertices). Each cube may be finite- or infinite-dimensional, and there may be infinitely many of them.

The usual notion of transition as an edge of a graph now gives way to that of a transition as any cell of non-zero dimension, with the dimension indicating the number of events participating jointly in that transition. A transition t may be said to run between its boundaries, the precise definition of which we only give for acyclic HDAs, which we now define.

[†] A very elegant characterisation of a cubical complex (which is beyond the scope of this paper) is as a functor $F : \mathbf{Bip} \to \mathbf{Set}$, where **Bip** is the category of bipointed sets or 0-0 algebras: algebras with two constants and no non-zeroary operations. The category of cubical complexes and their homomorphisms can then be taken to be the functor category $\mathbf{Set}^{\mathbf{Bip}}$, which is a topos by virtue of being a presheaf category.

2.10. *Acyclic HDAs and triadic event structures*

Higher dimensional automata seem at first sight orthogonal to event structures. Indeed, at the talk where the author first presented HDAs (Pratt 1991), Boris Trakhtenbrot asked from the front row at question time what relationship these higher dimensional automata had to event structures[†], for which we had no good answer. More recently, we have described (Pratt 2000) what we feel to be the correct connection, *via* acyclic HDAs and triadic event structures, which the present paper develops further. We sketch the essentials of this connection before proceeding to the main results of this paper.

In outline, the reconciliation of HDAs and event structures on A is accomplished by restricting the former while generalising the latter. HDAs are restricted to their acyclic case by defining them not as a set of cubes allowing sharing of cells, but rather as a subset of a single cube 3^A. And event structures are generalised from two-valued to three-valued states, dyadic to triadic, by expanding the set $2 = \{0, 1\}$ of possible values for membership of events in states to $3 = \{0, ⌐, 1\}$. The new state ⌐ is understood as *transition*, making our logical formulation three-valued, with ⌐ the 'edge' in what electrical engineers call edge-triggered logic connecting the 'levels' 0 and 1.

An ***acyclic HDA*** is a pair $\mathscr{A} = (A, X)$ where A is a set of events and X is a subset of 3^A whose elements are understood as the states of $\mathscr{A}$. In particular, the cube 3^A itself is the acyclic HDA $(A, 3^A)$ on A realising the independent or unrestricted occurrence of the events of A. Each A-tuple (configuration, state) x of 3^A is a cell whose dimension is the number of ⌐s in x and whose boundaries are all tuples in 3^A obtainable from x by setting one or more ⌐s in x to either 0 or 1. What makes 3^A acyclic is the linear ordering $0 <$ ⌐ < 1 on 3, which lifts to the natural partial ordering on 3^A in a way entirely analogously to that on 2^A.

A ***triadic event structure*** is defined exactly as for an acyclic HDA. Henceforth we use the terms interchangeably, preferring the latter when viewing the process geometrically.

The event a as the proposition 'a has occurred' now has three truth values: if a has not yet started, the truth value is 0; if a is ongoing or in transition, the truth value is ⌐; and if a has finished, its value becomes 1.

The removal of states from 3^A may be understood as furnishing 3^A with a certain structure, namely that structure that is disrespected by precisely the removed states. This is the view of a state as a morphism from A to 3 respecting whatever structure is imputed to A by the omission of certain states. While in many cases such structure may be independently characterised in more conventional ways (for example, with $\leqslant$ and $\#$ as with prime coherent event structures), this subsetting or 'sculpture' approach to specifying structure has three important benefits: simplicity, generality and categorical algebra. This last arises by formulating acyclic HDAs as triadic Chu spaces, inheriting their notion of morphism $f : (A, X) \to (B, Y)$ as a function $f : A \to B$ that is continuous in the sense that for all $y \in Y$, $y \circ f \in X$ ($- \circ f$ being the inverse-image function f^{-1}). For a more detailed account of this perspective, visit `http://chu.stanford.edu/`.

[†] More precisely, he asked about the relationship to event spaces, a variant of event structures we had recently introduced.

Triadic event structures make it very easy to distinguish $a||b$ from atomic mutual exclusion $ab + ba$. The difference is that only the former admits the state ⎍⎍ ($a =$ ⎍, $b =$ ⎍). In all other respects they are the same.

Although this distinction is a central benefit of the intermediate state, there are also other phenomena not expressible with two states that become expressible with three, for example, asymmetric conflict and numeric semaphores.

Asymmetric conflict. Asymmetric conflict is the condition that if a has happened, b cannot happen. Two-valued logic admits only the interpretation that the state ⎍⎍ ($a =$ ⎍, $b =$ ⎍) is disallowed, which automatically makes this condition symmetric. However, three-valued logic admits the meaning that the state 1⎍ is disallowed. That is, the actual occurrence of b ($b =$ ⎍ as opposed to its completion $b = 1$) is ruled out by the completion of a. If b happens first, taking us to 01, a can still happen afterwards *via* ⎍1 then 11, a route that avoids the forbidden 1⎍.

Strong asymmetric conflict. This is asymmetric conflict with the additional condition of mutual exclusion, that is, state ⎍⎍ is proscribed. This begins to look like sequential composition ba, but differs from it in that a without b is permitted. In fact, strong asymmetric conflict is expressible in basic process algebra as $a + ba$, and so is not needed as a primitive, unlike its weaker cousin. Whereas dyadic event structures identify all three of $a + ba$, $ab + ba$, and $a||b$, triadic event structures distinguish all three.

Numeric Semaphores. When n processes are competing for m resources, for example three children taking turns riding two ponies, it is natural to regulate the processes with a semaphore initialised to m. Each process obtains one of the resources from the pool by decrementing the semaphore, and returns it to the pool by incrementing it. The requirement that the semaphore remain non-negative at all times limits allocation to the available resources.

This system is representable with triadic event structures by the requirement that the number of ⎍'s (events in transition) in a state be at most m, or from the HDA perspective that the dimension not exceed m. The 3-children/2-ponies example has the three events of each child taking a turn riding a pony. There are a total of 27 possible configurations, and the limit of two ponies rules out just one of these, the one in which all three children are in the intermediate state of riding a pony. Geometrically, we have the surface of a cube, whose dimension is the number of ponies. For want of a pony, the interior is lost. We may write this process in 3-2 logic as $\neg(\widehat{a} \wedge \widehat{b} \wedge \widehat{c})$. Pushing the negation down expands this to $\overline{a} \vee a \vee \overline{b} \vee b \vee \overline{c} \vee c$, which expresses the requirement that the process must confine its activities to the six faces of the cube.

No change is needed to the definitions of the four process algebra operations in order to accommodate this new intermediate state, since only sequence and choice assume anything about K, and they look only for initial and final states; ⎍ is neither. However, the insertion of that additional state permits a reasonable notion of a 'step' of a computation, which is to be defined, as already discussed. And it considerably increases the number of process states for multi-event processes, as illustrated in Table 2 with the same examples as before.

Table 2. *Three-valued semantics of concurrency.*

a	a 0⌐2	$ba+ca$	a 000⌐100⌐1 b 0⌐1110000 c 00000⌐111
$a\|\|b$	a 0⌐10⌐10⌐1 b 000⌐⌐⌐111	$a(b+c)$	a 0⌐11111 b 000⌐100 c 00000⌐1
ab	a 0⌐111 b 000⌐1	$ab+ac$	a 0⌐11111 b 000⌐100 c 00000⌐1
$a+b$	a 0⌐100 b 000⌐1	$ab+ba$	a 0⌐1010⌐1 b 000⌐⌐111
$a\|\|(b+c)$	a 0⌐10⌐10⌐10⌐10⌐1 b 000⌐⌐⌐111000000 c 000000000⌐⌐⌐111	$ab\otimes cd$	ac 0⌐11111111111 ad 00000⌐⌐⌐11111 bc 000⌐10⌐10⌐111 bd 00000000000⌐1
$(b+c)a$	a 000⌐100⌐1 b 0⌐1110000 c 00000⌐111	$(a+b)\otimes(c+d)$	ac 0000000⌐10000⌐⌐11 ad 00000⌐100⌐⌐110000 bc 000⌐10000⌐1⌐10000 bd 0⌐10000000000⌐1⌐1

The transitional state distinguishes $a||b$ from $ab+ba$ according to whether or not, respectively, state ⌐⌐ is present. We still have $a(b+c) = ab+ac$ however.

The thirteen states of $ab \otimes cd$ constitute the thirteen elements of Allen's basic interval algebra (Allen 1984; Pratt 2000). And the states 0⌐⌐0 and ⌐00⌐ of $(a+b)\otimes(c+d)$ represent both pigeons in the process of simultaneously entering their respective holes.

2.11. *Step and run*

Triadic event structures permit a notion of computation that is both richer and more refined than for their dyadic counterparts. We define a ***step*** to be a pair (x, y) of distinct elements of 3^A (not necessarily in X) such that every coordinate of (x, y) is one of: 0, 0; 0, ⌐; ⌐, ⌐; ⌐, 1; or 1, 1 (that is, is equal or adjacent in Figure 1(b)) – thus (00, 0⌐), (00, ⌐⌐), (⌐⌐, 11), (0⌐, 01) and (0⌐, ⌐1) are all steps, but not (0⌐, 0⌐), (0⌐, 00) or (00, 01). The ***step graph*** of (A, X) is the graph with vertex set X and edge set all steps (x, y) with $x, y \in X$. A ***run*** of an acyclic HDA (A, X) is a path in its step graph.

The example steps (00, ⌐⌐) and (0⌐, ⌐1) suggest the following notions, which illustrate the richness possible with triadic event structure computations. A step (x, z) (and hence any run containing that step) ***bypasses*** a state y when (x, y) and (y, z) are both steps. A run ***synchronises*** events a and b when $a \equiv b$ at every state of the run; such a synchronising run thus bypasses all states in which a and b have 'drifted out of synch'. Event a ***segues*** into event b in a run when that run includes at least one state with $a =$ ⌐, $b = 0$, and another with $a = 1$, $b =$ ⌐, but bypasses all states in which $a = 1$, $b = 0$. Seguing can be thought of as a staggered form of synchronisation: the termination of a is synchronised with the starting of b.

Disallowing $(0, 1)$ as a step makes concurrent computational complexity a meaningful notion for triadic event structures. In the dyadic case, the transitions were just inclusions, and hence closed under composition, in which case, only sequential computational complexity, where one counts the total number of events performed without acknowledging the performance benefit of concurrency, makes sense. (One could allow credit for independence of participating events, but what is the right notion of independence here, and how should partial independence be weighted?) In the triadic case, steps are, in general, not closed under composition (bypassing provides only 'local' composition), and hence more than one step may be needed to accomplish a given run. In particular, ab takes at least three steps: a starts, then b starts (with a possible step in between in which a stops before b starts), and finally b stops; in general $a_1a_2\ldots a_n$ takes $n+1$ steps when all intermediate quiescent states are bypassed, and $2n$ steps when they are not. However, $a_1||a_2||\ldots||a_n$ can be accomplished in two steps, start and stop, provided they all stay in synch, though it can also be as slow as $a_1a_2\ldots a_n$ when there is no synchronisation at all (the run confines itself to the edges of 3^A, thereby losing the performance benefit of the higher dimensional cells.)

These notions of step and run can be described very efficiently for all four atomic automata in Figure 1. Treat all of them as reflexive graphs whose non-identity edges are drawn explicitly, and oriented to point upwards, and whose identity edges are left undrawn as understood. Do not, however, view these figures as the Hasse diagram of a poset, that is, do not take transitive closure. This is unavoidable for 1(a) and 1(c) because there are no paths of length 2, which would be needed to violate transitivity, and so they are automatically Hasse diagrams, but for 1(b) and 1(d) it takes two steps to get from 0 to 1.

We can in this way succinctly define the step graph on A to be the irreflexive hull of the product graph formed by multiplying A copies of the reflexive graph K, that is, K^A less the identity steps, which we have disallowed only to avoid meaninglessly long runs. Whereas the step graphs produced in this way from the graphs in 1(a) and 1(c) are transitive (since the product of transitive graphs is transitive), those from 1(b) and 1(d) are not transitive, and contain shortest-paths of length up to $2|A|$.

A run is then a path in the step graph K^A, and a run of a process is a maximal path in X, which is now understood as the intersection of the step graph with the states of the process. The advantage of this viewpoint is its independence of the details of K, which can be added to as required in order to accommodate notions of behaviour that are beyond the scope of the present paper, such as exceptions and aborting.

3. Results

In this section we state and prove two elementary results about triadic event structures or acyclic HDAs, concerning, respectively, the necessity of three values for distinguishing concurrent from interleaving behaviour (their sufficiency being implicit in our discussion of their use), and the basis problem for a concrete language of the associated three-valued logic. We then reassign the transitional state to a different role, that of *cancellation*, as a semantics for branching time. We conclude with a four-valued logic combining concurrency and branching time, having as its four values *before*, *during*, *after* and *cancel*, which are coded as 0, $\lrcorner\!\ulcorner$, 1 and $\times$, respectively.

3.1. *Necessity of three-valued event structures*

The above connection between triadic event structures and acyclic HDAs (Pratt 2000) demonstrates the sufficiency of three values, leaving open whether there might be some other approach that did not require us to abandon two-valued event structures. We felt intuitively that the passage to three values was not capricious but necessary, but have made no attempt to bring that intuition closer to the surface. Here we give a precise sense in which this passage is necessary.

Although a single bit cannot express much, sufficiently many bits can express any picture, concept, or fact that can be found in cyberspace or on a hard drive. Since any bit string of length 2^n can be expressed as the truth table of a Boolean formula in n atoms, any negative result about the expressive power of Boolean formulas must necessarily depend on assumptions about how the formulas are used. And if some obstacle is overcome by passing from two values to three, those assumptions must include accounting carefully for every bit, or it will be possible to refute the result by allowing two bits (binary digits) to simulate a trit (ternary digit).

We make the following assumptions. First, we assume that each event is associated to a single atom.

This assumption is popularly violated by the encoding of a 'long' event a as a pair of 'short' events, namely the start and end of a, which admits the possibility that a has started but not yet ended. To avoid the opposite possibility, one adds the restriction that if a has ended, it must have (previously) started. This eliminates one of the four assignments or states possible for two atoms, which boils this encoding down to exactly the three-valued case.

Three-valued logic can thus be understood as a two-bit start-end encoding of events in which one of the four configurations of two bits is disallowed. Viewing this situation as three values for one atom instead of two values for each of two atoms has several benefits. It simplifies language by halving the number of Boolean atoms. It simplifies logic by replacing the global axiom start-precedes-end with a cardinality adjustment to the set of truth values. And it exposes the underlying geometry of concurrency by turning disagreement of start and end into dimension of state: dimension 0 for agreement (with *before*, the event has neither started nor ended, while with *after*, the event has both started and ended), dimension 1 for disagreement (with *during*, the event has started but not ended).

The complete independence of $a\|b$ makes it a concurrent activity. The implicit mutual exclusion of $ab+ba$ is naturally understood as a choice followed by sequential behaviour. Event structures would not appear able to draw this distinction.

However, the one-atom–one-event assumption has a loophole: if we cannot duplicate atoms, let us duplicate the events themselves. Treat the events in the ab choice as being distinct from those in the ba choice, and call the latter $b'a'$ to make this distinction explicit. Then $ab+b'a'$ contains four events, while its automaton contains five states, $\{\}$, $\{a\}$, $\{a,b\}$, $\{a'\}$, and $\{a',b'\}$.

A drawback of this approach to distinguishing the concurrent behaviour of $a\|b$ from the interleaving behaviour of $ab+ba$ is that labelling now becomes necessary if any connection between a and a' is to be made. Otherwise the claimed distinction is really between $a\|b$ and $ab + cd$, a distinction not between concurrency and interleaving but simply between activities performing different tasks.

But labelling used in this way conflicts with the use of labelling to indicate when two unambiguously distinct events perform the same action. For example, aa consists of two consecutive occurrences of the action a, whereas the two appearances of a in $ab + ba$ are of the same occurrence in different temporal relationships to b. The labelling mechanism should be reserved for truly distinct occurrences of the same action.

Another drawback is that the automaton cannot forget the order in which ab or ba was made. Ideally, there would be a state $\{a, b\}$ reached by performing a and b sequentially in either order, and then forgetting the order as being immaterial. Treating as distinct the events associated with each order necessarily creates a permanent record of the choice because the two branches of the choice then arrive at distinct states.

With these concerns in mind, we make a second assumption: the distinction between interleaving and concurrency already arises for the unlabelled case of event structures. With this assumption, we can no longer meaningfully clone an event of an event structure as above.

The obvious correspondences famously confuse $a\|b$ with $ab + ba$. This, however, leaves open the possibility that event structures as a two-valued logic of event occurrences have some other interpretation that makes the desired distinction. Our theorem is the stronger result that concurrency–interleaving confusion must occur no matter how event structures are interpreted, subject only to the above two assumptions: events have only two values and identity across choices is maintained extensionally at the event level (as opposed to intensionally with the aid of labels).

The first result of this paper comes in two parts: the first for prime coherent event structures and the second for general event structures.

Theorem 8. Prime coherent event structures cannot distinguish interleaving from concurrent behaviour.

Proof. It suffices to consider $ab + ba$ and $a\|b$ themselves. Both evidently call for symmetric event structures, but there are only two such on two events: independence (no forbidden configurations) and conflict ($\{a, b\}$ forbidden). But we can hardly say a and b are performed concurrently if one of them does not happen, and in any event, conflict is assigned a separate meaning. This leaves independence as the only possibility. It can serve to denote either concurrent behaviour or interleaving or both, but it cannot distinguish the two notions. □

Now, event structures are of limited expressive power compared to general Boolean logic. The question then arises as to whether the latter has sufficient additional room to permit the concurrent behaviour $a\|b$ to be represented by some formula that is available by virtue of representing no interleaving behaviour.

Here a reasonable candidate presents itself, namely $a \equiv b$, which was unavailable with prime coherent event structures. This describes so-called *step* concurrency in which concurrent events occur synchronously: starting and ending together. (This is a different notion of step from the 0-to-$\ulcorner$, $\ulcorner$-to-1 notion of step used in this paper, since $\ulcorner$ does not enter into step concurrency.) We must now confront what we expect of $a\|b$. If we are satisfied with step concurrency, then, since logical formulas cannot confuse synchronised activity with interleaving activity, it follows that concurrent activity cannot be confused with interleaving either.

But if $a\|b$ is understood as permitting a to precede b, meaning that there can be a state in which a has happened and b has not, then it must be distinct from $a \equiv b$, as the latter insists on simultaneity, which allow only the state in which neither are done and the state in which both are done.

Without making any commitment to one interpretation or another, we can, nevertheless, ask whether it is possible to distinguish between the interleaving behaviour $ab+ba$ (atomic mutual exclusion), $a \equiv b$ (synchronisation) and $a\|b$ (independence). We show here that there are not enough dyadic event structures to draw all three distinctions: one of them must be either omitted or identified with one of the other two.

Theorem 9. Dyadic event structures cannot distinguish between all three of interleaving, synchronous and independent behaviour.

Proof. As before, we take as our counter-example the case of two events for which we wish to draw these distinctions. We have already argued that concurrent behaviour must accept the configuration $a = b = 1$ in order that both events happen. Dual reasoning forces acceptance of $a = b = 0$, since without that configuration one of a or b cannot happen during the behaviour, because 'happening' entails passing from state 0 to state 1. This leaves only two symmetric Boolean operations, *true* and $a \equiv b$, which are not enough for a three-way distinction. □

Synchronous behaviour should of course be associated with $a \equiv b$, and the interleaving behaviour $ab + ba$ clearly should not. We have the choice of identifying independent behaviour with either synchronous behaviour or interleaving behaviour, but there are not enough event structures to keep all three separate. If it is acceptable to identify concurrent with synchronised behaviour, then two-valued logic can satisfactorily realise $\|$ as $\equiv$. If not, then two values are not enough, that is, three values are necessary, and we have the necessary justification for the 3-2 logic of concurrency.

3.2. *A basis for 3-2 logic*

The passage from two logical values to three is invariably assumed to apply to both the inputs and outputs of logical operations. This follows a long tradition of simplicity through homogeneity: n-ary mathematical operations on a set K are customarily taken to be of type $K^n \to K$, even for such operations as division whose type would be more appropriately $K \times K^{\neq 0} \to K$.

For three-valued logic this tradition suggests taking the appropriate abstract Boolean formulas to be some if not all of the A-ary operations on 3, that is, functions $3^A \to 3$. And indeed, this has been the standard practice in all the explorations of three-valued logic of which we are aware, with the semantics of a binary operation such as $a \to b$ being given as a 3×3 truth table with entries drawn from 3.

However natural this homogeneity may seem, the set 3^{3^A} is considerably larger than we need to describe and reason about concurrent behaviour. For that purpose, and arguably for other situations calling for a logic with three-valued atoms, the smaller set 2^{3^A}, that is, functions $3^A \to 2$, is adequate. And at least as importantly, it fits perfectly with acyclic HDAs, triadic event structures and triadic Chu spaces.

Extending the practice of referring to logics whose operations are of type k^{k^A} as k-valued logics, we shall refer to logics whose operations are of type k^{j^A} as j-k-valued logics, or just j-k logics, with $j = 3, k = 2$ for the case at hand.

In practice, one works not with the whole of k^{j^A} but rather with just a few basic operations from which the remainder are then obtained by composition. For $j = k = 2$, ordinary Boolean logic, Scheffer stroke or NAND forms a complete basis for A non-empty, though the traditionally preferred basis is $\wedge, \vee, \neg$, or $\bigwedge, \bigvee, \neg$ when A is infinite.

In fact this preferred basis remains complete even under the restriction that $\neg$ be applied only to atoms. This follows from the completeness of the unrestricted basis (however proved) by the method of pushing negations down to the atoms with the help of the De Morgan laws: $\overline{P \vee Q} = \overline{P} \wedge \overline{Q}$ and its dual.

However, the completeness of this restricted case can also be proved *ab initio* by expressing the operation $f : 2^A \to 2$ as a disjunction $f = \bigvee_{f(t)=1} g_t$, each of whose disjuncts denotes an operation $g_t : 2^A \to 2$ that is 1 at exactly one A-tuple t of 2^A. Each disjunct g_t is in turn expressed as a conjunction of literals (possibly negated atoms), with one for each $a \in A$, and taking the sign of the a-th literal to be positive if $t_a = 1$ and otherwise negative. In fact, this is an excellent way of proving completeness of the unrestricted basis (and circumvents the De Morgan laws into the bargain.)

An equivalent syntax for literals writes a as $a = 1$ and $\overline{a}$ as $a = 0$. For fixed a and i, the formula $a = i$ is made an element of $2^A \to 2$ by defining it as $\lambda t.t_a = i$, the input to which is an A-tuple t whose a-th component t_a is to be compared with i. The literals then consist of all such elements of $2^A \to 2$, which form a set isomorphic to $A \times 2$.

This technique extends immediately to j-2 logic by allowing i to range over j values instead of 2. The set of such literals is then isomorphic to $A \times j$, with the obvious choice of name for the (a, i)-th literal being simply $a = i$. (Although we have been writing 3 for $\{0, \lrcorner, 1\}$, in the present context it is more natural to take j to be $\{0, 1, \ldots, j-1\}$.)

Theorem 10. The operations $\bigwedge$, $\bigvee$, and the literals $a = i$ form a complete basis for j-2 logic.

Proof. Every A-ary j-2 operation $f : j^A \to 2$ is evidently representable as

$$f = \bigvee_{f(t)=1} \bigwedge_{a \in A} a = t_a.$$

□

For any given $t \in j^A$, the function $\bigwedge_{a \in A} a = t_a$ from j^A to 2 is 1 at t and 0 elsewhere. $\bigvee_{f(t)=1}$ enumerates those t at which f is 1. When $j = 2$ this construction forms the complete disjunctive normal form (DNF) of f, where 'complete' means that every $a \in A$ appears with the appropriate sign in every needed disjunct. Evidently there is nothing special about the choice of 2 for j in this notion of normal form.

For 3-2 logic it suffices to expand any complete basis L for 2-2 (ordinary Boolean) logic to $L{\_\!\Gamma}$ by adjoining the unary operation $\widehat{a}$ understood as the literal $a = {\_\!\Gamma}$ (reverting to $3 = \{0, {\_\!\Gamma}, 1\}$). $L{\_\!\Gamma}$ thus has three forms of literal: negative $\overline{a}$, ***transitional*** $\widehat{a}$, and positive a.

The simplicity of this expansion is one of the benefits of 3-2 logic over homogeneous 3-3 logic, which requires operations other than ordinary conjunction and disjunction in order to generate all three output values.

A more practical basis allows literals to be combined with other Boolean operations besides $\bigwedge$ and $\bigvee$, allowing atomic mutual exclusion $ab + ba$, for example, to be expressed as $\neg(\widehat{a} \wedge \widehat{b})$. Pushing this negation down to the literals blows this formula up to $\overline{a} \vee a \vee \overline{b} \vee b$, a blow-up not encountered with 2-2 logic. This blown-up formula, which geometrically confines the process to the four sides of the $ab+ba$ square, makes clear why atomic mutual exclusion is not distinguishable from $a||b$ using dyadic event structures: it expresses a dyadic tautology but not a triadic one, whereas $a||b$ is both a dyadic and triadic tautology.

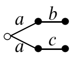
Though we shall not need it here, Chu spaces being adequately served by j-2 logic, the above construction for the j-2 case generalises to j-k with the reinterpretation of $\bigwedge$ and $\bigvee$ as max and min, and the addition of suitable constants and a selection operation.

Theorem 11. The operations $\bigwedge$ understood as max, $\bigvee$ as min, selection $m * n$ defined by $m * 1 = m$, $m * n = 0$ for $n \neq 1$, the literals $a = i$, and the constants $0, 1, \ldots, k-1$ form a complete basis for j-k logic.

Proof. Represent f as

$$\bigvee_{f(t)=m} m * \bigwedge_{a \in A} a = t_a. \qquad \square$$

3.3. *Branching time: 3-2 logic for the unlabelled case*

The basic example  of early branching appears to require event labels to relate the two transitions leaving the start state. Now, with $ab + ba$, the two occurrences of a were understood as being the same event in two alternatives, and likewise those of b. In the present situation we have two occurrences of a in $ab+ac$ that we would like to understand as being the *same* event in two alternatives differing only in their choice of whether b or c happens. Making this connection with labels again conflicts with the use of labels to account for the string **aa** as two *distinct* occurrences of the same action **a**, thus entailing two events, which is the same argument we made when considering concurrency. Then we were able to make the two occurrences of a in $ab + ba$ the same event *via* three-valued event structures; can we do the same here?

We shall show how this can be done with 3-2 logic. We will neglect concurrency for the time being.

We take the three event values to be *before*, *after* and *cancel*, which we code as 0, 1 and $\times$, respectively, as in Figure 1(c), with literals $\overline{a}$, a and $\not{a}$, respectively, defining the language $L_\times$ by analogy with $L{\_\!\Gamma}$. Each event starts out in state 0, and passes to one of 1 or $\times$ depending, respectively, on whether it happens or is cancelled. Thus 0 is the initial state of an event, and $\times$ and 1 are its two final states. Since event structures of this kind are structurally different from acyclic HDAs or triadic event structures, we shall call them ***cancelling event structures***.

A process is in its initial state when all its events are in the initial state, as with the two-valued case. However, we can now say that it is in a final state when each of its events is in one of the two possible final states. Without the $\times$ state we had no comparably simple and local test of finality: an event could be in state 0 either because it had been cancelled or because it had not yet got around to starting. The best we could do in that case was to declare a state to be final if it was not included in any larger state. Thus one benefit of $\times$ is that it permits a local definition of final state.

The process algebra operations for cancelling event structures are as for dyadic event structures with the following changes.

We first define $\checkmark\mathscr{A}$ more locally as $\mathscr{A}$ with the added requirement that every event be either completed (in state 1) or cancelled (in state $\times$). This is equivalent to saying that $\checkmark\mathscr{A}$ is obtained from $\mathscr{A}$ by removing all states in which at least one event is 0 (or $\lrcorner$ when that is included). We can now simplify $\checkmark\mathscr{A}$ in the definition of sequence to $\checkmark[A]$, defined as the process on A consisting of all possible final states (of which there are 2^A since every event is optionally cancellable). (The square brackets in $\checkmark[A]$ indicate that $\checkmark$ is applied to each member of its set argument, with the results then conjoined.) This simplification is possible because the definition of sequence independently imposes the constraint of being a state of $\mathscr{A}$. (The second-order notion of final state rules out this simplification, as does the popular method of explicitly labelling every state according to whether it is final, as is done with traditional automata theory.)

Sequence. Now we could keep our original definition of sequence $\mathscr{A}\mathscr{B}$, modified to $\mathscr{A} \wedge \mathscr{B} \wedge (B = 0 \vee A = \checkmark)$ with the above trick for termination, but then we still only have transitivity of sequence, rather than idempotence. Instead, we shall free up $\mathscr{A}\mathscr{B}$ a little in the case that there are any shared events, but not as much as with $B - A = 0 \ \vee\ \checkmark[A]$.

We define $\mathscr{A}\mathscr{B}$ to be $\mathscr{A} \wedge \mathscr{B} \wedge (B - A = 0 \ \vee\ \checkmark[A]) \wedge (B = 0 \ \vee\ \checkmark[A - B])$.

A little algebra shows this to be equivalent to $\mathscr{A} \wedge \mathscr{B} \wedge (B = 0 \ \vee\ \checkmark[A] \ \vee\ (B - A = 0 \wedge \checkmark[A - B])$, that is, the original definition weakened to liberate events common to $\mathscr{A}$ and $\mathscr{B}$ provided the rest of $\mathscr{A}$ has finished and the rest of $\mathscr{B}$ has not yet started.

For A and B disjoint, we have the ordinary notion of sequence. For $A = B$, however, we have $\mathscr{A}\mathscr{A} = \mathscr{A} \wedge \mathscr{A} = \mathscr{A}$ (since all events are common and therefore liberated), and therefore we have idempotence. But the symmetry of the definition also overcomes the counter-example $a(ba) \neq (ab)a$ that we encountered with only $B - A = 0 \ \vee\ \checkmark[A]$; both sides now denote $a \equiv b$. A little more algebra demonstrates associativity (exercise). As always, sequence is not at all commutative.

Thus we have the surprising result that sequence behaves conventionally with disjoint arguments (the normal case in automata theory), but, nevertheless, it is an idempotent operation!

The operation is now sufficiently rational to allow us to define a new internal implication, which we write as $\mathscr{A} \to \mathscr{B}$, defined to be the process algebra operation $\mathscr{B}\mathscr{A}$. (To avoid conflict with material (Boolean) implication, the latter can be written $\mathscr{A} \supset \mathscr{B}$.)

We have not dared to explore the consequences of this notion based on the definition of sequence we were using for dyadic event structures because of its awkward notion of termination. This implication depends in an essential way on having cancellation as an alternative termination state to 1. However, in the absence of conflict, that is, when termination requires all events to happen, $\mathscr{A}\mathscr{B}$ simplifies to just $\mathscr{A}||\mathscr{B}$, with the additional requirement that every event of A precede every event of B, which can be written

$$\mathscr{A} \wedge \mathscr{B} \wedge \bigwedge_{a\in A, b\in B} (b \to a),$$

(making this case of $\mathscr{A}\mathscr{B}$ clearly associative). This simpler notion can itself be taken as a novel internal implication $\mathscr{B} \to \mathscr{A}$ for Boolean logic. The difference between this simpler notion of implication and $\mathscr{A}\mathscr{B}$ in the case of cancelling event structures must be kept in mind when $\mathscr{A}$ contains any conflict $\times$, since then not every event of $\mathscr{A}$ can happen, and thus in this simplified $\mathscr{B} \to \mathscr{A}$, $\mathscr{B}$ never gets to start (every satisfying assignment assigns 0 to all of B not in A), yet does get to start in $\mathscr{A}\mathscr{B}$ provided $\mathscr{A}$ has at least one final state. This simpler notion also gives a direction to start out from in the above exercise for associativity of sequence.

Choice. In the dyadic case ($K = \{0,1\}$), we defined $\mathscr{A} + \mathscr{B}$ as

$$A \cup B = 0 \ \vee \ (\mathscr{A} \neq 0 \ \wedge \ B - A = 0) \ \vee \ (\mathscr{B} \neq 0 \ \wedge \ A - B = 0).$$

The significance of $B - A = 0$ in dyadic choice is that when $\mathscr{A}$ is chosen over $\mathscr{B}$, those events of B not participating in $\mathscr{A}$ can never pass to 1 (or even to $\lrcorner$ when we reintroduce it). The difficulty here is that one cannot deduce this 'never' from inspection of individual states: one must consider the whole process.

The $\times$ value lets us express the 'never' part of this explicitly in a single state: a cancelled event is one that can never move towards 1. Thus, when the process enters a state in which $\mathscr{A}$ is chosen, the events of $\mathscr{B}$ that are not shared with $\mathscr{A}$ are all explicitly cancelled *in that state*, instead of merely being left unstarted, and *vice versa* when $\mathscr{B}$ is chosen. Once cancelled, an event stays cancelled, because there is no transition from $\times$ to any other value of K in the primitive event automata of Figures 1(c) and 1(d).

We therefore substitute $B - A = \times$ for $B - A = 0$ and $\mathscr{A} \nleq \times$ for $\mathscr{A} \neq 0$ (to ensure that some event of A starts as soon as the choice is made) in the dyadic definition of choice to yield the definition

$$A \cup B = 0 \ \vee \ (\mathscr{A} \nleq \times \ \wedge \ B - A = \times) \ \vee \ (\mathscr{B} \nleq \times \ \wedge \ A - B = \times).$$

(However naturally this definition may appear to have flowed from the dyadic definition, it was, in fact, suggested to us by P S Thiagarajan as an alternative to the definition we had previously derived in exactly the same way from an equivalent definition of dyadic choice as $A \cup B = 0 \ \vee \ (\mathscr{A} \ \wedge \ B - A = 0) \ \vee \ (\mathscr{B} \ \wedge \ A - B = 0)$. When $B - A = \times$ is substituted here for $B - A = 0$, we obtain a notion of choice in which the cancellations are performed *before* starting the selected process. Unfortunately, our definition fails associativity, as witnessed by the state $\times 00$, which is present in $a + (b + c)$ but absent from $(a + b) + c$. In the Thiagarajan definition the chosen process starts up simultaneously with the cancellation of all rejected events, *including those cancelled in the selected process at its outset*. Hence, when c is chosen in $a + (b + c)$, a and b are cancelled simultaneously, thus precluding state $\times00$. Viewed more abstractly, the disjuncts in our definition of dyadic choice were not disjoint; substituting $B - A = \times$ for $B - A = 0$ preserves associativity provided the disjuncts are disjoint.)

Other notions of choice. An alternative definition of $\mathscr{A} + \mathscr{B}$ allows the chosen process to start without waiting for the unchosen events to be cancelled. They would eventually be cancelled due to the termination requirement that every event either terminate or be cancelled. This, however, furnishes $ab + ac$ with a state 100, giving it the possibility of deferring the choice of b or c to the same time $a(b + c)$ chooses. The two processes are, nevertheless, distinguished by the *possibility* in $ab + ac$ of cancelling an event before a has happened. With this definition, we have $\mathscr{A}(\mathscr{B} + \mathscr{C}) \vdash \mathscr{A}\mathscr{B} + \mathscr{A}\mathscr{C}$.

However, $\mathscr{A}\mathscr{B} + \mathscr{A}\mathscr{C} \vdash \mathscr{A}(\mathscr{B} + \mathscr{C})$ makes more intuitive sense, meaning that $\mathscr{A}\mathscr{B} + \mathscr{A}\mathscr{C}$ is more constrained (has fewer options) than $\mathscr{A}(\mathscr{B} + \mathscr{C})$. This is achievable as follows. Whenever a run of $\mathscr{A} + \mathscr{B}$ (as defined by Thiagarajan) passes through x and then y, such that a is 0 in x and $\times$ in y, we adjoin to the states of $\mathscr{A} + \mathscr{B}$ the additional state x' derived from x by changing b from 0 to $\times$. This definition satisfies $\mathscr{A}\mathscr{B} + \mathscr{A}\mathscr{C} \vdash \mathscr{A}(\mathscr{B} + \mathscr{C})$ while retaining the crucial distinction that $a(b + c)$ contains 100 and so *may* perform a before cancelling either b or c, unlike $ab + ac$.

All definitions of choice proposed above, buggy or not, are easily seen to be consistent with dyadic choice in the sense that they reduce to it under the identification $\times = 0$.

The Thiagarajan definition of choice being expressible in closed form, unlike the two alternatives proposed above making $\mathscr{A}\mathscr{B} + \mathscr{A}\mathscr{C}$ and $\mathscr{A}(\mathscr{B} + \mathscr{C})$ comparable (at least as we have expressed them), we take it as the meaning of choice in what follows.

By convention, the atomic process a happens, that is, may not be cancelled when standing alone, and therefore, by default, is permitted only the states 0 and 1. Thus the logical form of a is no longer *true* but rather $\neg \not{a}$, which is equivalent to $\overline{a} \vee a$. This also implies that even though $\checkmark\mathscr{A}$ now allows $\times$ in final states, $\checkmark a$ still means $a = 1$.

Substituting $\mathscr{A} = a$ and $\mathscr{B} = b$ in the preferred definition of choice, $a + b$ translates into logic as

$$(\overline{a} \wedge \overline{b}) \vee (a \not\leqslant 0 \wedge \not{b}) \vee (b \not\leqslant 0 \wedge \not{a}).$$

This asserts that $a + b$ has a zero state, and that when it is not in that state, one of a or b is cancelled and the other has started (which in the absence of $\lrcorner\!\ulcorner$ means it is done). That is,

$$a + b = \begin{array}{r|c|} \cline{2-2} a & 01\times \\ b & 0\times1 \\ \cline{2-2} \end{array}$$

Event a can be turned into a cancellable event by adding the empty process 0[†] that has no events and one state to yield the process $a + 0$ having one event and all three possible states. For $a + 0$ to take the 0 branch means that a is cancelled and nothing else

[†] 0 denotes the zeroary Boolean operation (constant) tt, which has no events and one state; the Boolean constant $f\!f$ is the logical form of the process $\bullet$ with no events and no states.

Table 3. *Three-valued semantics of branching time.*

a	$\begin{array}{l} a \end{array}\boxed{\begin{array}{l} 01 \end{array}}$	$ba + ca$	$\begin{array}{l} a \\ b \\ c \end{array}\boxed{\begin{array}{l} 00101 \\ 011\times\times \\ 0\times\times 11 \end{array}}$
$a \Vert b$	$\begin{array}{l} a \\ b \end{array}\boxed{\begin{array}{l} 0101 \\ 0011 \end{array}}$	$a(b + c)$	$\begin{array}{l} a \\ b \\ c \end{array}\boxed{\begin{array}{l} 0111 \\ 001\times \\ 00\times 1 \end{array}}$
ab	$\begin{array}{l} a \\ b \end{array}\boxed{\begin{array}{l} 011 \\ 001 \end{array}}$	$ab + ac$	$\begin{array}{l} a \\ b \\ c \end{array}\boxed{\begin{array}{l} 01111 \\ 001\times\times \\ 0\times\times 01 \end{array}}$
$a + b$	$\begin{array}{l} a \\ b \end{array}\boxed{\begin{array}{l} 01\times \\ 0\times 1 \end{array}}$	$ab + ba$	$\begin{array}{l} a \\ b \end{array}\boxed{\begin{array}{l} 0101 \\ 0011 \end{array}}$
$a \Vert (b + c)$	$\begin{array}{l} a \\ b \\ c \end{array}\boxed{\begin{array}{l} 010101 \\ 0011\times\times \\ 00\times\times 11 \end{array}}$	$ab \otimes cd$	$\begin{array}{l} ac \\ ad \\ bc \\ bd \end{array}\boxed{\begin{array}{l} 011111 \\ 000111 \\ 001011 \\ 000001 \end{array}}$
$(b + c)a$	$\begin{array}{l} a \\ b \\ c \end{array}\boxed{\begin{array}{l} 00101 \\ 011\times\times \\ 0\times\times 11 \end{array}}$	$(a + b) \otimes (c + d)$	$\begin{array}{l} ac \\ ad \\ bc \\ bd \end{array}\boxed{\begin{array}{l} 01\times \\ 0\times 1 \\ 0\times 1 \\ 01\times \end{array}}$

happens. Hence the equation $\mathscr{A} + \mathscr{A} = \mathscr{A} + 0$ holding for dyadic event structures fails for the triadic case, because with $\mathscr{A} + \mathscr{A}$ the two branches have the same set of events and hence no events need cancel (unless they already did so in $\mathscr{A}$). We do, however, have idempotence $\mathscr{A} + \mathscr{A} = \mathscr{A}$ when $\mathscr{A}$ has the all-zero state.

So states 0 and $\times$ participate in the definitions of both sequence and choice, while 1 enters in the definition of sequence only. Concurrence and orthocurrence remain independent of any particular elements of K.

With the new atomic state $\times$, our examples change as in Table 3.

The process $a + b$ starts out in the 00 start, then one of a or b is cancelled and the other proceeds. An alternative semantics would be to omit the initial 00 state and make the choice of a or b at the outset by starting out in one of two initial states: either $0\times$ or $\times 0$. The role of the common initial 00 state is to allow $a + b$ to defer the choice as long as there is other work to be done beforehand, as in $c(a + b)$. It also helps ensure that every process built with these connectives from atoms has a unique initial state.

In $a(b + c)$ we see the significance of the 00 state for $b + c$, which permits a to finish before choosing one of b or c. In $ab + ac$, by contrast, a cannot proceed until the choice of b or c has been made, and in that respect differs from $a(b + c)$.

In the absence of $\lrcorner\!\ulcorner$, $ab + ba$ remains equal to $a \Vert b$. Choice cannot distinguish them because even though $ab + ba$ must make a choice, no event is thereby cancelled. It follows that $\times$ can only distinguish early from late choice when that choice affects *which* events get done. More subtle influences, for example, on order of events, cannot be distinguished by $\times$. In particular, $a(bc + cb)$ and $abc + acb$ remain the same, namely,

$$\begin{array}{l} a \\ b \\ c \end{array}\boxed{\begin{array}{l} 01111 \\ 00101 \\ 00011 \end{array}}.$$

The $\times$ state does not enter with any of concurrence, sequence or orthocurrence, which give the same results as in Table 1.

$(a+b) \otimes (c+d)$ is now described in more detail than in the previous two tables. In particular, we can see that when (a, c) is cancelled, so is (b, d): no pigeon may start to enter its hole until it has been determined which pigeon/hole pairs have been ruled out. But the Thiagarajan definition of choice calls for the pigeons to enter the holes simultaneously with the cancellations.

The following observation concerning $\surd\mathscr{A}$, describing $\mathscr{A}$ at its termination (that is, the deletion from $\mathscr{A}$ of all states in which any event is still 0 without, however, changing the event set A), allows for a tidier account of pigeons if not trains. This theorem depends on the first-order definition of $\surd\mathscr{A}$ made possible by $\times$; we have previously seen pigeonhole counter-examples ($a||b$ *vs.* $0+a+b+a||b$) to this theorem with our $K=\{0,1\}$ definition of $\surd\mathscr{A}$.

Theorem 12. $\surd(\mathscr{A} \otimes \mathscr{B}) = \surd\mathscr{A} \otimes \surd\mathscr{B}$.

Proof. $\surd(\mathscr{A} \otimes \mathscr{B}) \vdash \surd\mathscr{A} \otimes \surd\mathscr{B}$ because every state of $\surd(\mathscr{A} \otimes \mathscr{B})$ is free of zeroes and hence can be built solely from zero-free states of $\mathscr{A}$ and $\mathscr{B}$, that is, from states of $\surd\mathscr{A}$ and $\surd\mathscr{B}$. Conversely, $\surd\mathscr{A} \otimes \surd\mathscr{B} \vdash \surd(\mathscr{A} \otimes \mathscr{B})$ because $\surd\mathscr{A} \otimes \surd\mathscr{B} \vdash \mathscr{A} \otimes \mathscr{B}$, and no state of $\mathscr{A} \otimes \mathscr{B}$ with a 0 can be produced using only states of $\surd\mathscr{A}$ and $\surd\mathscr{B}$. $\square$

In particular, $\surd(a+b) \otimes \surd(c+d)$ more efficiently describes the two ways of putting two pigeons in two holes. The case of three pigeons and two holes, with up to two pigeons per hole, has six solutions, which we might expect to be described by $\surd((a||b)+(b||c)+(c||a)) \otimes \surd(c+d)$. The relevant calculations for these two examples are as follows:

$$\begin{array}{r|cc|}\hline a & \times & 1 \\ b & 1 & \times \\ \hline\end{array} \otimes \begin{array}{r|cc|}\hline c & \times & 1 \\ d & 1 & \times \\ \hline\end{array} = \begin{array}{r|cc|}\hline ac & 1 & \times \\ ad & \times & 1 \\ bc & \times & 1 \\ bd & 1 & \times \\ \hline\end{array} \qquad \begin{array}{r|ccc|}\hline a & \times & 1 & 1 \\ b & 1 & \times & 1 \\ c & 1 & 1 & \times \\ \hline\end{array} \otimes \begin{array}{r|cc|}\hline d & \times & 1 \\ e & 1 & \times \\ \hline\end{array} = \begin{array}{r||}ad \\ ae \\ bd \\ be \\ cd \\ ce\end{array}$$

While $\surd(a+b) \otimes \surd(c+d)$ is as expected, $\surd((a||b)+(b||c)+(c||a)) \otimes \surd(c+d)$ has no states! The problem is that the new definition of final state means that $((a||b)+(b||c)+(c||a)) \otimes (c+d)$ has no final states. This comes about because in the hole with only one pigeon, we always have a second pigeon in state 0 when finality demands it be in state $\times$. We have been sloppy in writing $(a||b)+(b||c)+(c||a)$ when what we really meant was $a+b+c+(a||b)+(b||c)+(c||a)$, the choice of one pigeon or two. Back when we were not distinguishing 0 from $\times$ there was no difference. Now that we are making the distinction, we have to count pigeons more accurately. Adding in the $a+b+c$ possibility gives the expected six solutions, as follows:

$$\begin{array}{r|cccccc|}\hline a & 1 & \times & \times & \times & 1 & 1 \\ b & \times & 1 & \times & 1 & \times & 1 \\ c & \times & \times & 1 & 1 & 1 & \times \\ \hline\end{array} \otimes \begin{array}{r|cc|}\hline d & \times & 1 \\ e & 1 & \times \\ \hline\end{array} = \begin{array}{r|cccccc|}\hline ad & 1 & \times & \times & 1 & \times & 1 \\ ae & \times & 1 & 1 & \times & 1 & \times \\ bd & \times & 1 & 1 & \times & \times & 1 \\ be & 1 & \times & \times & 1 & 1 & \times \\ cd & \times & 1 & \times & 1 & 1 & \times \\ ce & 1 & \times & 1 & \times & \times & 1 \\ \hline\end{array}$$

A similar calculation shows that the example $(a||b) \otimes (c+d)$ of up to two pigeons in each of two holes must be changed to $(0+a+b+a||b) \otimes (c+d)$, where 0 is the consistent (one-state) zero-event process.

This points up yet another problem with the intensional (pre-cancellation) definition of final state, one that admittedly only arises *via* orthocurrence with the four-operator process algebra presented here, but that could easily arise when the enthusiastic process algebraist replaces orthocurrence by some other operator more to their liking. The problem is that a state that should not be final (for example, because some programmer pact has decreed states of its ilk to be temporarily inconsistent system states) might under the old definition of finality be unjustly promoted to the rank of final state by the disappearance of certain crucial later states that the pact does recognise as consistent. This might fool the process whose turn is next into forging ahead when it is dangerous to do so. The second-order definition of finality can be trickier to administer than the first-order one made possible by explicit cancellation.

On the face of it, we appear to have used the same 3-2 logic for branching time as for concurrency. However, there is a crucial difference in how the third value enters. For concurrency we identified a neglected intermediate state through which each event passes on its way from *before* to *after*. This additional state makes the natural relationship between 2 and 3 one of inclusion: 2 is a subset of 3. This is reflected in the passage back to a two-valued semantics of concurrency, namely by *deleting* those states containing any occurrence of ⌐, the process by which we recover Table 1 from Table 2 (and Table 3 from Table 4, yet to come).

For branching time, however, we *refined* the *before* state according to whether it was willing to pass to the *after* state. This relationship between 2 and 3 thus makes 2 a quotient of 3 resulting from the identification $\times = 0$. Performing this quotient by replacing $\times$ by 0 turns Table 3 into Table 1 (and Table 4 into Table 2) not by deleting states, but by *identifying* those states that previously differed only by having $\times$ in place of 0. One side effect of this quotient is to identify $a(b+c)$ and $ab+ac$, which the $\times$ state nicely distinguishes, without, however, disturbing the generally accepted equality between $(b+c)a$ and $ba+ca$, in both of which the choice is made at the beginning.

Despite this basic difference between these two uses of 3-2 logic, we adopt the same logic for both. The only difference is in how we think of the third literal: as transition or cancellation.

3.4. *Infinite processes*

In the case of an infinite process, one with infinitely many events, it is reasonable to suppose, at least for an effective process, that each of its final states have only finitely many events in the 1 state. For example, the sequential process $\mathbf{a}^*$ that performs finitely many instances of action **a** and then halts is described as the process on the infinite set $A = \{a_0, a_1, a_2, \ldots\}$ that changes each of $a_0, a_1, \ldots$ in turn from 0 to 1, and then at some finite stage changes the remaining events to $\times$. The allowed non-final states are therefore all sequences of the form 1^*0^ω, while the allowed final states are all sequences of the form $1^*\times^\omega$ (non-final and final being the only two possibilities). Going by our definition of step, any non-final state has a step to a final state that changes a finite initial segment of the trailing 0^ω to 1's (the stragglers) and the rest to $\times$. We will revisit this example after bringing in ⌐ to join $\times$, whose principle benefit is a more rational notion of step.

The process $(\mathbf{ab})^*$ is treated as for $\mathbf{a}^*$ with two differences. First, we replace the set of final states with sequences of the form $(11)^*\times^\omega$, that is, only an even number of 1s is allowed in final states. Second, we now need a labelling function, though this is a triviality: just label all the events alternately **a** and **b**. From this point of view, labelling is always a triviality, and all the interesting structure resides in the unlabelled part of the theory.

The process $(\mathbf{a}+\mathbf{b})^*$ has an event set better described as $A = \{a_0, a_1, \ldots\} \times \{a, b\} = \{(a_0, a), (a_0, b), (a_1, a), (a_1, b), \ldots\}$. Every state has an initial segment of length $k \geqslant 0$ in which every pair of events $(a_i, a), (a_i, b)$ for $i < k$ are either $1\times$ or $\times 1$ according to whether a or b, respectively, was chosen at the i-th stage. Immediately after this segment there is an optional $(a_k, a), (a_k, b)$ pair that is either $0\times$ or $\times 0$, indicating that one of a or b has been chosen but has not yet happened. Past that point there are two possibilities: either every pair is 00, or, provided every chosen event has happened (no $0\times$ or $\times 0$ pair), every pair is $\times\times$, the latter constituting the final states. These are all the states of $(\mathbf{a}+\mathbf{b})^*$. The operational interpretation should be clear, with the same caveat as before that any state has a single step to a final state, which is dealt with as before by ⎍. A variant of this process would allow the tail to be all $\times\times$ even in the presence of one $0\times$ or $\times 0$ pair, which corresponds to the process deciding it is time to halt without thereby entering a final state (the last chosen event still needs to be done, though without ⎍, the concluding step would have the opportunity to cancel the chosen event instead of performing it).

3.5. *Combining branching time and concurrency*

We now have two applications of 3-2 logic: one to concurrency and one to branching time. However, their third value has a different meaning in each, so we cannot use them together. Instead, we combine them by passing to 4-2 logic, in which *before* and *after* are common to all fragments, but *during* and *instead-of* are distinguished as separate states originating in the concurrency and branching time worlds, respectively. The result is the primitive event automaton of Figure 1(d).

The transitions of our four-state automaton (assumed to be oriented upwards, making 0 the start state) induce transitions between states of a multi-event process coordinatewise: one state vector can pass to another just when the passage is permitted in each coordinate. Thus there is a transition from 00 to $0\times$, and one from 00 to ⎍$\times$ expressing the simultaneous stepping of two events, one of which is cancelled, but no transition from $0\times$ to 0⎍ because an event may not start once it is cancelled.

It is tempting to allow a transition from ⎍ to $\times$ as well, but the semantics of cancellation should forbid starting in order to avoid unwanted side effects. The passage from ⎍ to an alternative to 1 should instead be to a state called *abort*, a fifth state with the implication that the aborted event may have had some effect before its untimely end. Adding this state to the above setup should require no modification other than to include the abort state among the final states of an event. An alternative here would be to treat it instead as an error state, in which case a state vector would be an error state when at least one event had aborted. We leave this to future treatments.

Theorem 10 ensures that 4-2 logic has a complete basis just as for 3-2 logic, namely the infinitary monotone connectives $\bigwedge$ and $\bigvee$ (and any other Boolean operations that might be convenient to include, and which may be viewed as either expansions to the signature or abbreviations) together with the four literals $\bar{a}$, $\widehat{a}$, a, and $\not{a}$ constituting the language $L_{⎍\times}$.

Table 4. *Four-valued process algebra.*

a	*a* 0⌐1	$ba+ca$	*a* 000⌐100⌐1 *b* 0⌐111×××× *c* 0××××⌐111
$a\|\|b$	*a* 0⌐10⌐10⌐1 *b* 000⌐⌐⌐111	$a(b+c)$	*a* 0⌐11111 *b* 000⌐1×× *c* 000××⌐1
ab	*a* 0⌐111 *b* 000⌐1	$ab+ac$	*a* 0⌐111⌐111 *b* 000⌐1×××× *c* 0××××00⌐1
$a+b$	*a* 0⌐1×× *b* 0××⌐1	$ab+ba$	*a* 0⌐1010⌐1 *b* 000⌐⌐111
$a\|\|(b+c)$	*a* 0⌐10⌐10⌐10⌐10⌐1 *b* 000⌐⌐⌐111×××××× *c* 000××××××⌐⌐⌐111	$ab \otimes cd$	*ac* 0⌐11111111111 *ad* 00000⌐⌐⌐11111 *bc* 000⌐10⌐10⌐111 *bd* 00000000000⌐1
$(b+c)a$	*a* 000⌐100⌐1 *b* 0⌐111×××× *c* 0××××⌐111	$(a+b)\otimes(c+d)$	*ac* 0⌐⌐11×××× *ad* 0××××⌐⌐11 *bc* 0××××⌐1⌐1 *bd* 0⌐1⌐1××××

In this combined semantics, the definitions of the process algebra connectives given for branching time require no change. The only other change needed is that the atomic event a is defined as *a*$\boxed{0⌐1}$, as for the 3-2 logic of concurrency. (Thus a process with an event in state ⌐ that is not permitted to pass to state 1 cannot terminate; that event is deemed to be in a blocked state. One might instead suppose it to be in a divergent state, but on the reasonable and popular assumption that any atomic event may on closer inspection turn out to be compound (refinement), blocking and divergence must be indistinguishable, to which anyone who has patiently awaited a web page can attest.)

The same behaviours we considered before expand as in Table 4.

Note that $ab+ba$ coincided with $a||b$ in the case of branching time semantics but not when concurrency semantics was added. We still have $(b+c)a = ba+ca$, as expected, but not $a(b+c) = ab+ac$.

The principle example in this paper of an infinite process, $\mathbf{a}^*$ in Section 3.4, had the property with × but without ⌐ that every non-final state had a step to a final state in which finitely many stragglers finished and the rest are cancelled. This situation, which clearly does not capture the essence of $\mathbf{a}^*$, would not have arisen in any reasonable sequential model. It arose in our ⌐-less concurrent model for lack of any reasonable way to acknowledge the performance contribution of intermediate degrees of independence.

We capitalise on ⌐ by adding to the states of $\mathbf{a}^*$ a third class of states: those of the form 1^*⌐0^ω in which one event is in progress. With the more reasonable step semantics we use with ⌐, it is still possible for any stragglers to make the jump to the 1 state as part of the step from a penultimate to a final state, but every 0 in a penultimate state must change to × in the final step because the semantics of a step no longer allows a direct leap from 0 to 1. Hence stragglers must have already started by the time of the final jump, but now there is only room for at most one such straggler. Whether there is a straggler depends on whether the non-final state from which that last step is taken has zero or one ⌐'s, that is, is of dimension zero or one.

3.6. *HDAX geometry*

Although the geometric view of an acyclic HDA (A, X) with $X \subseteq 3^A$ understood as a sculpted cube is clear enough, how should a subset of 4^A, or HDAX[†] as we shall call it, be understood geometrically?

Given an HDAX state x that is $\times$ at a subset $B \subseteq A$ (that is, all events of B are cancelled in x), we propose to view x as a cancellation-free state of 3^{A-B}. An event that is cancelled in the course of a run thus drops out of sight altogether, which projects the computation as a whole onto a lower-dimensional space by projecting out the axis of the cancelled event. This projection does not change the state of the run at the time of projection, but merely removes some of the possible future directions.

Consider, for example, $(a||b) + a$, which is

$$\begin{array}{c|ccccccccccc|} \hline a & 0 & ⌐ & 1 & 0 & ⌐ & 1 & 0 & ⌐ & 1 & ⌐ & 1 \\ b & 0 & 0 & 0 & ⌐ & ⌐ & ⌐ & 1 & 1 & 1 & \times & \times \\ \hline \end{array}\,.$$

In the beginning its geometric meaning is indistinguishable from that of $a||b$, namely a square. Taking the second option of the choice by stepping from state 00 to state ⌐$\times$, however, projects the square onto the a axis, thereby removing b's opportunity to start. (It is always the future that is projected onto the present, never the past onto the present, because one can only cancel an event that has not yet started.) Note that a run of this process can take the step (⌐0, ⌐$\times$), that is, the existence of state 0$\times$ does not preclude the possibility that this state could be bypassed, allowing the collapse to happen while a is in progress, or even, *via* the step $(10, 1\times)$, after it terminates. Contrast this with $ab + ac$, in which no run can initiate a without first choosing between b and c. The reason $(a||b) + a$ need not commit to a branch before starting a is that the right-hand alternative a contains no cancellable states!

A run that terminates does so at the final (all-ones) corner of a cube of dimension the number of events that happened in the course of the computation. Normally, this number will be finite, even if the computation began with infinitely many dimensions. Structure in the final cube, in the form of omitted cells, constitutes the obstacles the run had to negotiate; with the full cube present, the run could have been as short as two steps, but the more typical case will omit many cells and the computation will accordingly have been obliged to take much longer.

This dynamic picture of HDAX behaviour could certainly be made static, but we do not see any way of doing this that is sufficiently geometrically intuitive as to recommend it over the dynamic picture.

As for other structures modelled as Chu spaces, HDAX morphisms are obtained simply as the morphisms of $\mathbf{Chu}_4$, the category of Chu spaces over a 4-letter alphabet. A Chu morphism can be understood as a continuous function, in that the inverse image of every state of its target is required to be a state of its source, where the inverse image of $y : B \to K$ under $f : A \to B$ is defined as the composite $y \circ f : A \to K$. The category of biextensional Chu spaces (extensional Chu spaces that are also separable – every pair of distinct events is distinguished by some state) over any given K is $*$-autonomous in the

[†] An alternative reading is: Hitherto, During, After, $\times$.

sense of Barr (1979), that is, a self-dual symmetric monoidal closed category, and, as such, a model of linear logic (Girard 1987).

Counterintuitively, $\mathbf{Chu}_4$ is defined independently of any structure on $K = 4$, which is taken to be just a set, and yet the Chu morphisms respect the structure on processes implied by the above transitions on $\{0, \lrcorner, 1, \times\}$. The key lies in the seemingly clairvoyant ability of $\mathbf{Chu}_K$'s morphisms to respect structure imposed *a posteriori* on K, a phenomenon we have explained elsewhere (Pratt 1999). The secret is that every possible structure on K lifts in a uniform and natural way to structure on objects of $\mathbf{Chu}_K$ in a way that ensures that the (preordained) morphisms respect precisely that structure, no more and no less. An early example of the phenomenon is Lafont and Streicher's observation (Lafont and Streicher 1991) that the Chu morphisms between vector spaces represented as Chu spaces (which they call games) over the *set* (not field) of complex numbers are exactly the linear functions between those vector spaces, despite the lack of any reference to complex arithmetic in the definition of that category of Chu spaces. Slightly less magically (since the Chu morphisms are defined as continuous functions), they also observe that the Chu morphisms between Chu spaces over $\{0, 1\}$ representing topological spaces are exactly the continuous functions between those spaces.

3.7. *Comparison with Rodriguez-Anger branching time*

All four structures of Figure 1, along with two others called partially ordered time and relativistic time, have their counterparts in Rodriguez and Anger's branching time variant (Rodriguez and Anger 2001) of Allen interval algebras (Allen 1984). The elements of an interval algebra are the possible relationships between the two intervals as they slide past each other on parallel tracks. The relationships, of which there are 13, are traditionally derived from the three binary relations $<$, $=$ and $>$ between two endpoints, with one from each interval. To analyse the case where the tracks diverge at some point, Rodriguez and Anger apply a fourth (symmetric) binary relation $||$ of incomparability, which holds just when the comparison is between points on distinct arms of the diverged tracks, and combine it with the other three relations to obtain the 19 interval relationships possible with diverging tracks.

This variant of interval algebra continues an ongoing study by Rodriguez and Anger of such variants based on the introduction of additional binary relations such as $||$. Referring to the original interval algebra as characterising linear time, they treat partially ordered time and relativistic time using 4 and 6 binary relations, respectively, to give 29 and 82 interval relationships, respectively (Anger and Rodriguez 1991; Rodriguez and Anger 1993a; Rodriguez and Anger 1993b). Partially ordered time corresponds to tracks that diverge only temporarily, with the same notion $||$ of incomparability as above, while relativistic time is the same with two additional relations describing one endpoint at the point of divergence or convergence with the other on the diverged section, which corresponds to the boundary of the relativistic light cone.

Elsewhere (Pratt 2000), we recast the relations $<$, $=$ and $>$ between pairs a, b as atomic states 0, $\lrcorner$, and [†] of such pairs construed as events to allow the relation algebra

[†] Our earlier HDA papers coded these three values as 0,1,2.

techniques typically used to study interval algebras to be replaced by direct application of the orthocurrence operator, thereby correlating the Allen configurations to the 13 states of the orthocurrence $ab \otimes cd$ as in Table 2 above. We then interpreted the additional relationships as additional event states and represented the two intervals as processes in such a way that their orthocurrence had the above 29 and 82 states, respectively.

In Rodriguez and Anger (2001), the authors point out that time is symmetric for all three of linear time, partially ordered time and relativistic time, and argue that this symmetry masks a need to reverse one of the arguments to orthocurrence when applied to interval algebras. To demonstrate this, they break the symmetry of partially ordered time by replacing temporary divergence of tracks by permanent divergence (no rejoining), naming the resulting notion of time branching time. Orthocurrence applied as in the other examples now gives the wrong answer, but time-reversing one of the intervals (by exchanging their endpoints and also exchanging $<$ and $>$ in the matrix) does give the correct answer.

While these names for various kinds of time are appropriate within the setting of interval algebras, the question naturally arises as to whether they have the same meaning as used elsewhere. In particular, can this interval-algebra notion of branching time distinguish $a(b+c)$ from $ab+ac$?

The role of the interval in the interval algebra analysis of time is to represent sequencing, as evident in the successful use of $ab \otimes cd$ in modelling interval algebra for the three previous notions of time. Now, when an interval slides off on a different track from the other interval, the permanent incomparability of its leading endpoint eventually results in permanent incomparability of its trailing endpoint. It follows that if $||$ is used to distinguish $a(b+c)$ from $ab+ac$ by treating it as the state of the unchosen event of $b+c$ in the manner of cancellation, any event following the unchosen one must itself enter state $||$. But this would mean that d in $a(b+c)d$ could not happen. More generally, any process containing a choice would block the first time it made any choice.

This establishes that $||$ cannot be used like $\times$ if it is to pass the litmus test for branching time of distinguishing $a(b+c)$ from $ab+ac$. This does not rule out the possibility of drawing this distinction using $||$ in some other role, for example, one that views the one instance of a in $a(b+c)$ as being comparable with itself while the two copies of a in $ab+ac$ are incomparable (by virtue of having decided differently between b and c). Such a role would then fully justify the name 'branching time' for this variant of interval algebra. We do not see, however, how to do this in the unlabelled event structure framework as used in the present paper.

This difference between $||$ and $\times$ emphasises the versatility of orthocurrence as a neutral operator that respects the different meanings of these states to give correct results for processes using either one (when applied correctly of course).

It is worth pointing out an alternative analysis of branching-time interval algebra in which orthocurrence need not reverse one of its arguments, which is consistent with not doing so for any other application of orthocurrence. The need for reversal arises when the processes ab and cd in $ab \otimes cd$ are treated as having the same orientation, following nearly two decades of interval algebra tradition. However, inspection of the trains-and-stations scenario reveals that if ab is an eastbound system of trains, placing train a to the east of train b, then stations must be encountered in the order west to east, placing station c west of station d in cd.

To capture divergence, we can replace the stationary stations c, d by a westbound pair of trains c, d moving along a second parallel track, with train c ahead of, and hence west of train d. Assuming the tracks diverge when traveling east, the westbound trains reverse the order in which $||$ is encountered: instead of blocking, they unblock. The proper analysis of orthocurrence $\mathscr{A} \otimes \mathscr{B}$ in this case is then seen to be that $\mathscr{B}$ is simply a different system of trains from $\mathscr{A}$, experiencing scenery isomorphic to $\mathscr{A}$'s but in the reverse order. In this analysis, the need to reverse one argument to orthocurrence becomes a property not of orthocurrence but of its constituent processes, much as the abstract group theoretic analysis of the integers views subtraction $i - j$ as merely a convenient shorthand for $i + (-j)$. Both notations are legitimate and interchangeable.

4. Concluding reflections

Our earlier aspersions on homogeneity in type notwithstanding, there may possibly be some merit in the homogeneous formulation of three-valued logic, that is, the type $3^A \to 3$, for either concurrency or branching time alone, or, conceivably, even the homogeneous formulation $4^A \to 4$ of four-valued logic when concurrency and branching time are combined (but this starts to look unwieldy). There might, for example, be some justification for expanding the number of information distinctions to match that of the number of temporal distinctions. Scott domains or information systems leap immediately to mind here, specifically, **Bool**, consisting of the two truth values tt and ff together with $\perp$ satisfying $\perp \sqsubseteq tt$ and $\perp \sqsubseteq ff$. Standardly interpreted, information systems live on the automaton side of the schedule/automaton or time/information duality, in that their elements are conventionally understood as states ordered by information content *via* $\sqsubseteq$, with the bottom element $\perp$ constituting the state of utter ignorance.

A domain-oriented model would allow three states of knowledge about HDA cells: present, absent and undecided, corresponding to the values tt, ff and $\perp$ of the 3-element CPO **Bool**. Such a model would lend itself to the treatment of *discovery* of HDAs, that is, the automatic production of concurrent automata *via* a suitable recursive procedure. The bottom state is total ignorance about all cells. The maximal states are the HDAs towards which such a procedure would progress starting from ignorance.

Is the homogeneity of the resulting $3^A \to 3$ type structural or merely superficial? In particular, does 3 have a natural structure here that is a counterpart to the rich Boolean structure of 2? And can the three temporal values *before*, *during* and *after* be linked in some way to the three information values tt, ff and *undecided*, for example, in terms of triadic Chu spaces?

Elsewhere (Casley *et al.* 1991), we have pointed out that there are just two commutative idempotent 3-element quantales, each corresponding to a natural use for three-valued time, with the one that is a Heyting algebra, which we called **3**, being the appropriate one for causal-accidental time, and the other, **3**$'$, the one for strict/non-strict or before/during/after time. Is **3**$'$ the appropriate quantale for the three-valued CPO for the type **Bool**? The only connection we see is that the so-called face 'lattice' of **3**$'$ as a 1-cell (made an actual lattice by adjoining a bottom element of dimension -1) is the order dual of the CPO **Bool**. However, we do not see any good structural connection between face lattice structure and quantale structure, which suggests that cardinality may be the only connection between the two threes in this approach to making sense of $3^A \to 3$. We leave this homogeneity question as an open problem possibly bearing further investigation.

A far more important problem is the comprehensive treatment of fixed points, solutions of $\mathscr{A} = \tau\mathscr{A}$ (as well as prefixed points $\tau\mathscr{A} \vdash \mathscr{A}$ and postfixed points $\mathscr{A} \vdash \tau\mathscr{A}$), whether least, greatest, optimal, *etc*. Here the natural choice of language to start out with for τ is (in our biased view, of course) the four basic process algebra operations defined in this paper. But process algebra operations are limited only by the imagination of process algebraists and the tolerance of users for languages allowed to grow like Topsy, leaving the subject of fixpoints for HDAXs dauntingly wide open.

References

Allen, J. F. (1984) Towards a general theory of action and time. *Artificial Intelligence* **23** 123–154.

Anger, F. D. and Rodriguez, R. V. (1991) Time, tense, and relativity revisited. In: Bouchon-Meunier, B., Yager, R. R. and Zadeh, L. A. (eds.) *Uncertainty in Knowledge Bases: Proc. of the 3rd International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, IPMU'90*, Springer-Verlag 286–295.

Baeten, J. C. M. and Weijland, W. P. (1990) *Process Algebra*, Cambridge University Press.

Barr, M. (1979) ∗-Autonomous categories. *Springer-Verlag Lecture Notes in Mathematics* **752**.

Barr, M. (1991) ∗-Autonomous categories and linear logic. *Mathematical Structures in Computer Science* **1** (2) 159–178.

Bergstra, J. A., Ponse, A. and Smolka, S. A. (eds.) (2000) *Handbook of Process Algebra*, Elsevier (North-Holland).

Bergstra, J. A. and Klop, J. W. (1984) Process algebra for synchronous communication. *Information and Control* **60** 109–137.

Brookes, S. D., Hoare, C. A. R. and Roscoe, A. D. (1984) A theory of communicating sequential processes. *Journal of the ACM* **31** (3) 560–599.

Buckland, R. and Johnson, M. (1996) Echidna: A system for manipulating explicit choice higher dimensional automata. In: *AMAST'96: Fifth Int. Conf. on Algebraic Methodology and Software Technology*, Munich.

Casley, R. T., Crew, R. F., Meseguer, J. and Pratt, V. R. (1991) Temporal structures. *Mathematical Structures in Computer Science* **1** (2) 179–213.

Fajstrup, L., Goubault, E. and Raussen, M. (1998) Detecting deadlocks in concurrent systems. In: Proc. of CONCUR'98. *Springer-Verlag Lecture Notes in Computer Science* **1466** 332–347.

Genrich, H. J. and Thiagarajan, P. S. (1984) A theory of bipolar synchronisation schemes. *Theoretical Computer Science* **30** 241–318.

Girard, J.-Y. (1987) Linear logic. *Theoretical Computer Science* **50** 1–102.

Gischer, J. L. (1988) The equational theory of pomsets. *Theoretical Computer Science* **61** 199–224.

van Glabbeek, R. (1991) Bisimulations for higher dimensional automata. (Manuscript available as `http://theory.stanford.edu/~rvg/hda`).

van Glabbeek, R. and Plotkin, G. (1995) Configuration structures. In: *Logic in Computer Science*, IEEE Computer Society 199–209.

van Glabbeek, R.J. and Vaandrager, F.W. (1987) Petri net models for algebraic theories of concurrency. In: Proc. PARLE, II. *Springer-Verlag Lecture Notes in Computer Science* **259** 224–242.

Goubault, E. (1993) Homology of higher-dimensional automata. In: Proc. of CONCUR'92, Stonybrook, New York. *Springer-Verlag Lecture Notes in Computer Science* **630** 254–268.

Goubault, E. (1995a) Schedulers as abstract interpretations of hda. In: *Proc. of PEPM'95*, La Jolla, ACM Press.

Goubault, E. (1995b) *The Geometry of Concurrency*, Ph.D. thesis, École Normale Supérieure.

Goubault, E. (1996a) Durations for truly-concurrent actions. In: *Proceedings of ESOP'96*, Springer-Verlag 173–187.

Goubault, E. (1996b) A semantic view on distributed computability and complexity. In: *Proceedings of the 3rd Theory and Formal Methods Section Workshop*, Imperial College Press.

Goubault, E. (ed.) (2000) Geometry and concurrency special issue (7 papers). *Mathematical Structures in Computer Science* **10** (4) 409–573.

Goubault, E. and Cridlig, R. (1993) Semantics and analysis of Linda-based languages. In: Proc. 3rd Int. Workshop on Static Analysis, Padova. *Springer-Verlag Lecture Notes in Computer Science* **724** 72–86.

Goubault, E. and Jensen, T.P. (1992) Homology of higher dimensional automata. In: Proc. of CONCUR'92, Stonybrook, New York. *Springer-Verlag Lecture Notes in Computer Science* **630** 254–268.

Grabowski, J. (1981) On partial languages. *Fundamenta Informaticae* **IV** (2) 427–498.

Gunawardena, J. (1994) Homotopy and concurrency. *EATCS Bulletin* **54** 184–193.

Gupta, V. (1994) *Chu Spaces: A Model of Concurrency*, Ph.D. thesis, Stanford University. (Tech. Report, available as `http://boole.stanford.edu/pub/gupthes.ps.gz`.)

Gupta, V. and Pratt, V.R. (1993) Gates accept concurrent behaviour. In: *Proc. 34th Ann. IEEE Symp. on Foundations of Comp. Sci.* 62–71.

Halmos, P.R. (1974) *Finite-Dimensional Vector Spaces*, Springer-Verlag.

Harel, D., Kozen, D. and Tiuryn, J. (2000) *Dynamic Logic*, MIT Press.

Lafont, Y. and Streicher, T. (1991) Games semantics for linear logic. In: *Proc. 6th Annual IEEE Symp. on Logic in Computer Science*, Amsterdam 43–49.

Mazurkiewicz, A. (1977) Concurrent program schemes and their interpretations. Technical Report DAIMI Report PB-78, Aarhus University, Aarhus.

Milner, R. (1980) A Calculus of Communicating Systems. *Springer-Verlag Lecture Notes in Computer Science* **92**.

Nielsen, M., Plotkin, G. and Winskel, G. (1981) Petri nets, event structures, and domains, part I. *Theoretical Computer Science* **13** 85–108.

Park, D. (1981) Concurrency and automata on infinite sequences. In: Proc. Theoretical Computer Science. *Springer-Verlag Lecture Notes in Computer Science* **104** 167–183.

Papadimitriou, C. (1986) *The Theory of Database Concurrency Control*, Computer Science Press.

Petri, C.A. (1962) Fundamentals of a theory of asynchronous information flow. In: *Proc. IFIP Congress 62, Munich*, North-Holland 386–390.

Plotkin, G.D. and Pratt, V.R. (1996) Teams can see pomsets. In: *Proc. Workshop on Partial Order Models in Verification*, AMS.

Pnueli, A. (1977) The temporal logic of programs. In: *18th IEEE Symposium on Foundations of Computer Science* 46–57.

Pratt, V.R. (1976) Semantical considerations on Floyd-Hoare logic. In: *Proc. 17th Ann. IEEE Symp. on Foundations of Comp. Sci.* 109–121.

Pratt, V. R. (1982) On the composition of processes. In: *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages.*

Pratt, V. R. (1985) Some constructions for order-theoretic models of concurrency. In: Proc. Conf. on Logics of Programs, Brooklyn. *Springer-Verlag Lecture Notes in Computer Science* **193** 269–283.

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

Pratt, V. R. (1991) Modeling concurrency with geometry. In: *Proc. 18th Ann. ACM Symposium on Principles of Programming Languages* 311–322.

Pratt, V. R. (1999) Chu spaces: Notes for school on category theory and applications. Technical report, University of Coimbra, Coimbra, Portugal. (Manuscript available as `http://boole.stanford.edu/pub/coimbra.ps.gz`.)

Pratt, V. R. (2000) Higher dimensional automata revisited. *Mathematical Structures in Computer Science* **10** 525–548.

Pratt, V. R. (2001) Orthocurrence as both interaction and observation. In: Rodriguez, R. V. and Anger, F. D. (eds.) *Proc. Workshop on Spatial and Temporal Reasoning, IJCAI'01*, Seattle.

Rodriguez, R. V. and Anger, F. D. (1993a) Constraint propagation + relativistic time = more reliable concurrent programs. In: *Proc. of the Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems IEA/AIE-93*, Edinburgh, Scotland 236–239.

Rodriguez, R. V. and Anger, F. D. (1993b) An analysis of the temporal relations of intervals on relativistic space-time. In: Bouchon-Meunier, B., Valverde, L. and Yager, R. R. (eds.) *IPMU'92: Advanced Methods in Artificial Intelligence – Proc. of the 4th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*, Springer-Verlag 139–148.

Rodriguez, R. V. and Anger, F. D. (2001) Branching time via Chu spaces. In: Rodriguez, R. V. and Anger, F. D. (eds.) *Proc. Workshop on Spatial and Temporal Reasoning, IJCAI'01*, Seattle.

Sassone, V. and Cattani, G. L. (1996) Higher-dimensional transition systems. In: *Proceedings of LICS'96.*

Shields, M. (1985) Deterministic asynchronous automata. In: Neuhold, E. J. and Chroust, G. (eds.) *Formal Models in Programming*, Elsevier.

Takayama, Y. (1996) Extraction of concurrent processes from higher-dimensional automata. In: *Proceedings of CAAP'96* 72–85.

Winskel, G. (1980) *Events in Computation*, Ph.D. thesis, Dept. of Computer Science, University of Edinburgh.

Winskel, G. (1982) Event structure semantics for CCS and related languages. In: Proc. 9th ICALP. *Springer-Verlag Lecture Notes in Computer Science* **140** 561–576.

Winskel, G. (1986) Event structures. In: Petri Nets: Applications and Relationships to Other Models of Concurrency – Advances in Petri Nets, Bad-Honnef. *Springer-Verlag Lecture Notes in Computer Science* **255**.

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

Winskel, G. (1988b) An introduction to event structures. In: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, REX'88, Noordwijkerhout. *Springer-Verlag Lecture Notes in Computer Science* **354**.