Electronic Notes in Theoretical Computer Science 83 (2004)
URL: http://www.elsevier.nl/locate/entcs/volume83.html 18 pages

Cumulative Computing

Yifeng Chen [1]

Department of Mathematics and Computer Science
University of Leicester
Leicester LE1 7RH, UK

Abstract

In this paper we use the concept of resource cumulation to model various forms of computation. The space of cumulations (called a *cumulator*) is simply represented as a five tuple consisting of a well-founded partial order, a monoid and a volume function. The volume function is introduced to simplify reasoning about limit points and other topological properties. A specification command is a set of cumulations. Typical phenomena of concurrency such as reactiveness, safety and liveness, fairness, real time and branching time naturally arise from the model. In order to support a programming theory, we introduce a specification language that incorporates sequentiality, nondeterminism, simple parallelism, negation and general recursions. A new fixpoint technique is used to model general recursions. The language is applied to the case study on CSP, which becomes a special model of cumulative computing with a combination of four resource cumulators of alphabet, termination, trace and refusal. All laws of cumulative computing are also valid for CSP and the generalization from CSP to Timed CSP can be achieved by simply combining the four cumulators with real time. Loops whose bodies may take zero time can then be modeled more satisfactorily.

1 Introduction

Many aspects of computing can be modeled as cumulation of resources. For example, a bubble-sorting algorithm takes $O(n^2)$ steps to terminate where cost is a resource consumed by the algorithm. In real-time computing, time is a kind of resource: a process 'consumes' non-negative amount of time. A computation may also produce resources. For example, a reactive process generates an increasingly longer sequence of intermediate states called trace.

We can think of resources more generally. A sequential program that successfully terminates in 10000 steps is as good as another program with the same result that terminates in 10 steps. However, a nonterminating program

[1] Email: `yc10@mcs.le.ac.uk`

is always different from any terminating program. We can treat *termination* as a 'resource': a nonterminating program consumes infinite resource of 'termination', while a terminating program consumes zero such resource.

Resource cumulation can be measured. For example, length is a good measure for the trace of a reactive process. A cumulation is said to be *infinite* if its measure is infinite. Infinite cumulation is a conceptual abstraction (or approximation) for resource exhaustion. The set of all possible cumulations of a resource forms a *cumulator*.

A specification is a statement on the observables of computation (or alternatively a set of cumulations). An observable is *cumulative* if it is related to some resource. Not all observable aspects of computation are cumulative. For example, the temperature of a physical system is not a resource but a 'state' that can change up and down over time. Whether or not an observable is cumulative depends on not only its nature but also the level of abstraction. For example, the number of instructions executed by a CPU indicates the cost of a program and thus corresponds to a cumulator. However if we intend to model the process of debugging, the execution can then be reversed, and the number of instructions only denotes the point of execution. That does not mean a debugging process consumes no resources — any back tracking at least takes time, which is a typical cumulative resource.

In this paper, we will use the concept of resource cumulation to model various forms of computation. A simple specification language called *cumulative computing* is introduced. Primitive commands become constant specifications, while program compositions become functions on specifications.

The notion of 'resources' is not new in computer science. For example, resource at a *place* in a Petri net represents a local state [14]. In operating systems, the availability of resources can be represented using semaphores [6]. Other aspects of resources such as static resource ownership have also been studied in logic [8,13]. In this paper, we focus on the cumulation (or dynamics) of resources. In this sense, our approach is closer to the studies of the dynamic behaviours of real-time and reactive processes [4,5,9,11,14]. Indeed they are the typical computational models we intend to study. Our formalism resembles domain theory in many aspects, although the use of volume function is new. It is introduced to simplify the reasoning about limit points and other topological properties.

Communicating Sequential Processes (or CSP) [9] is a language that incorporates reactiveness, sequentiality, deadlocks, livelocks, parallelism and general recursions. *Failures-divergences model* is a popular semantic model of CSP [16]. Hoare and He [10] presented the model in predicative semantics using a number of special variables to denote the observation on trace, termination, waiting and refusal set of events. These variables are in fact related to resources and will be typed as cumulators in this paper. Timed CSP [15,12] is a generalization of untimed CSP. Using cumulative computing, we can obtain Timed CSP by simply combining the cumulators of untimed CSP with real time. For example, the cumulator of termination is replaced by the cumulator of real time.

In Section 2, the notion of cumulator is formalized, and several cumulator constructors are introduced. In Section 3, specification commands and their algebraic laws are studied. The techniques will be applied to case studies on CSP and Timed CSP in Section 4.

2 Cumulators

A resource cumulator can be formalized as a five tuple.

Definition 1 *A five-tuple* $(X, \leqslant\, ; \ \mathbf{0},\ ^\frown\, ; \ |\cdot|)$ *is a cumulator, if*

(i) *the well-founded partial order* $(X, \leqslant)$ *is closed under non-empty glb;*
(ii) *the chop operation* $^\frown : X \times X \hookrightarrow X$ *is monotonic and associative, and for any* $x, y \in X$ *if* $x \leqslant y$ *then there exists* $z \in X$ *such that* $x^\frown z = y$*;*
(iii) *the monoid* $(X, \mathbf{0},\ ^\frown)$ *satisfies* $\mathbf{0}^\frown x = x^\frown\mathbf{0} = x$ *and* $\mathbf{0} \leqslant x$ *for any* x*;*
(iv) *the volume function* $|\cdot| : X \to [0,\infty]$ *is monotonic and strict, i.e.* $|\mathbf{0}| = 0$.

Note that the chop operation may be a partial function in which case its correspondence with the partial order must still hold.

Partial order and monoid have been well studied in mathematics and widely applied in computer science. The unusual part of our definition is the volume function. A volume function measures the amount of resource cumulation. With such additional information we can then reason about the dynamics of resource cumulation. For example, a resource is exhausted when its volume reaches infinity ∞. Another example is the comparison of the speeds of two different cumulators: to synchronize a reactive process with a running clock, we simply require that the volume (or length) of the process equals the time shown on the clock.

Example 2.1

(i) The cumulator $\mathit{RTime} \mathrel{\widehat{=}} ([0,\infty], \leqslant\, ; \ 0, + \ ; \ \mathit{id})$ represents relative cumulation of real time, while $\mathit{AbsTime} \mathrel{\widehat{=}} ([0,\infty], \leqslant\, ; \ 0, \max \ ; \ \mathit{id})$ represents absolute cumulation of real time.
(ii) The cumulator $\mathit{Clock} \mathrel{\widehat{=}} (\mathbb{N}_\infty, \leqslant\, ; \ 0, + \ ; \ \mathit{id})$ is an integer clock.
(iii) The cumulator $\mathit{Trace}(T) \mathrel{\widehat{=}} (T^{*\infty}, \leqslant\, ; \ \langle\,\rangle,\ ^\wedge \ ; \ |\cdot|)$ represents the trace of a reactive process where $T^{*\infty}$ is the set of all sequences (including the ω-infinite ones) of events from T. The order $s \leqslant t$ holds iff s is a prefix of t. $s\,^\wedge t$ denotes the concatenation of the two sequences. If s is an infinite sequence, then for any t, $s\,^\wedge t = s$. $|s|$ denotes the length of s, and s_i denotes the i-th element of the sequence s where $0 \leqslant i < |s|$. The restriction $s \upharpoonright A$ of a trace s by a set A is a trace including only the elements from A. The merge $s \interleave t$ is a set of traces that are interleavings of the two traces s and t.

(iv) The cumulator *String*$(A) \mathrel{\widehat{=}} (A^*,\ \preccurlyeq\ ;\ \text{“”},\ \max\ ;\ |\cdot|)$ represents strings, each of which is a finite sequence of characters from the alphabet A. $s \preccurlyeq t$ holds iff s is less than t in lexical order. $\max(s,t)$ identifies the greater string in lexical order. $|s|$ denotes the length of a string s. The length of the empty string “” is 0. For example we have “Yifeng”$\succcurlyeq$“Jeff” and “Chen”$\preccurlyeq$“Sanders”.

(v) The cumulator *Optimization* $\mathrel{\widehat{=}} ([0,\infty],\ \geqslant\ ;\ \infty,\ \min\ ;\ 1/x)$ represents an optimization process. We assume that $1/0 = \infty$ and $1/\infty = 0$. We use the variable x to denote the argument of a function. Thus $1/x$ is the same as $\lambda x \cdot 1/x$.

(vi) The cumulator *Termin* $\mathrel{\widehat{=}} (\{\mathtt{true}, \mathtt{false}\},\ \Leftarrow\ ;\ \mathtt{true},\ \wedge\ ;\ |\cdot|)$ represents termination/nontermination. We let `true` denote termination or ‘no resource consumption’ and let `false` denote nontermination or ‘resource exhaustion’. Thus $|\mathtt{true}| = 0$ and $|\mathtt{false}| = \infty$.

(vii) The cumulator *Set*$(T) \mathrel{\widehat{=}} (\mathcal{P}(T),\ \subseteq\ ;\ \emptyset,\ \cup\ ;\ \mathtt{card})$ represents the power set of a set T. We assume that the cardinality $\mathtt{card}(S)$ of a finite set S is the number of elements in it. If S is any infinite set, then $\mathtt{card}(S) = \infty$. The cumulator *Set*$'(T) \mathrel{\widehat{=}} (\mathcal{P}(T),\ \supseteq\ ;\ T,\ \cap\ ;\ |\cdot|)$ also represents the power set of a set T, although the subsets are ordered by set containment, the concatenation is set intersection, and the volume is the number of elements not in the set $|S| \mathrel{\widehat{=}} \mathtt{card}(T \setminus S)$.

A cumulator is *normal* if no infinite cumulation can be further extended, i.e. $|x| = \infty$ and $x \preccurlyeq y$ implies $x = y$. All above cumulators are normal. Composite cumulators can be constructed from simple ones. Let $C \mathrel{\widehat{=}} (X,\ \leqslant\ ;\ \mathbf{0},\ ^\frown\ ;\ |\cdot|)$ and $C' \mathrel{\widehat{=}} (X',\ \leqslant'\ ;\ \mathbf{0}',\ ^{\frown'}\ ;\ |\cdot|')$ be two cumulators.

Definition 2 (Cartesian product) *The Cartesian product of C and C' is defined by $C \times C' \mathrel{\widehat{=}} (X \times X',\ \leqslant''\ ;\ (\mathbf{0}, \mathbf{0}'),\ ^{\frown''}\ ;\ |\cdot|'')$ where*

$$\begin{aligned}
(x,x') \leqslant'' (y,y') &\mathrel{\widehat{=}} x \leqslant y \wedge x' \leqslant' y' \\
(x,x') {}^{\frown''} (y,y') &\mathrel{\widehat{=}} (x {}^\frown y,\ x' {}^{\frown'} y') \\
|(x,x')|'' &\mathrel{\widehat{=}} \max(|x|,\ |x'|)\ .
\end{aligned}$$

Example 2.2 The Cartesian-product cumulator (*Clock* $\times$ *Clock*) is a pair of independent clocks that run freely at their own speeds.

Definition 3 (Restriction) $C \restriction Y \mathrel{\widehat{=}} (X \cap Y,\ \leqslant \cap (Y \times Y)\ ;\ \mathbf{0},\ ^{\frown''}\ ;\ |\cdot|)$ *is a restricted cumulator if it is a cumulator. Here we assume that $x {}^{\frown''} y \mathrel{\widehat{=}} x {}^\frown y$ if $x, y \in Y$ and $x {}^\frown y \in Y$.*

Example 2.3 The restricted cumulator (*Clock* $\times$ *Clock*)$\restriction \leqslant$ represents a pair of clocks between which the first one never runs faster than the second.

A finitely-restricted cumulator is a cumulator restricted by the set of finite cumulations.

Definition 4 (Finite restriction) *The finite restriction of the cumulator C is defined by $Fin(C) \ \widehat{=}\ C \restriction \{\, x \in X \mid |x| < \infty \,\}$.*

Example 2.4 (i) An integer counter is a finitely-restricted clock $Counter \ \widehat{=}\ Fin(Clock)$.

(ii) A *semaphore* is a protected non-negative integer variable accessible to only the operations 'wait' and 'signal' besides initialization [6]. The number of 'wait' operations executed must be bounded by the initial value plus the number of 'signal' operations executed. A semaphore corresponds to a restricted cumulator $(Counter \times Counter) \restriction R$ where $x \, R \, y \ \widehat{=}\ (x \leqslant I + y)$. The first counter and the second counter count the numbers of executed 'wait' and 'signal' operations respectively, while I denotes the initial value of the semaphore.

There are many kinds of fairness in the literature [7]. All of them require 'fair' distribution of resources among competing consumers such that no resource can be kept cumulating at the expense of another resource's exhaustion.

Definition 5 (Fair product) *The fair product of C and C' is defined by $C \otimes C' \ \widehat{=}\ (C \times C') \restriction \approx$ where $x \approx y \ \widehat{=}\ (|x| = \infty) \Leftrightarrow (|y|' = \infty)$.*

Example 2.5 The fair product $(Clock \otimes Clock)$ is a pair of clocks running fairly. If one of them stops, the other must stop in finite steps.

Definition 6 (Synchronous product) *The synchronous product of C and C' is defined by $C \bullet C' \ \widehat{=}\ (C \times C') \restriction \approx$ where $x \approx y \ \widehat{=}\ |x| = |y|'$.*

Example 2.6 The synchronous product $(Clock \bullet Clock)$ is a pair of synchronized clocks running at the same speed.

The cumulation of one resource may have priority over the cumulation of another resource.

Definition 7 (Priority product) *The priority-product of C and C' is defined by $C \ltimes C' \ \widehat{=}\ (X \times X', \leqslant'' ; (\mathbf{0}, \mathbf{0}'), \frown'' ; |\cdot|'')$ where*

$$
\begin{aligned}
|(x, x')|'' &\ \widehat{=}\ \max(|x|, |x'|) \\
(x, x') \frown'' (y, y') &\ \widehat{=}\ (x, x' \frown' y') \lhd (x = x \frown y) \rhd (x \frown y, y') \\
(x, x') \leqslant'' (y, y') &\ \widehat{=}\ x < y \ \vee \ (x = y \wedge x' \leqslant' y'),
\end{aligned}
$$

and $A \lhd b \rhd B$ denotes "if b then A else B".

Example 2.7 The cumulator $(Surname \ltimes Firstname)$ represents a name list in which names are primarily ordered by surnames and then by first names. Here we assume $Surname = Firstname \ \widehat{=}\ String$. For example we have ("Chen","Yifeng")$\preccurlyeq$("Sanders","Jeff") where $\preccurlyeq$ is the order of the combined cumulator.

In most applications, resources are bounded. For example, a system may fail if it does not terminate before timeout.

Definition 8 (Overflow) *The overflow cumulator of C over a set Y of limit points is defined by* $C \nearrow Y \ \widehat{=}\ (X,\ \leqslant\ ;\ \mathbf{0},\ {}^\frown\ ;\ |\cdot|'')$ *where* $\mathbf{0} \notin Y$ *and* $|x|'' \ \widehat{=}\ \infty \lhd (\exists y \in Y \cdot y \leqslant x) \rhd |x|$.

Example 2.8 (i) The overflow cumulator $\mathit{RTime} \nearrow \{t\}$ represents a real-time process with a timeout limit t.

(ii) The overflow cumulator $\mathit{Optimization} \nearrow \{\epsilon\}$ represents an optimization process with an accuracy criterion ϵ.

The cumulation of one resource may depend on the availability of another resource.

Definition 9 (Control product) *The control-product of C and C' is defined by* $C \rhd C' \ \widehat{=}\ (X \times X',\ \leqslant''\ ;\ (\mathbf{0}, \mathbf{0}'),\ {}^{\frown''}\ ;\ |\cdot|'')$ *where*

$$\begin{array}{rcl} |(x,x')|'' & \widehat{=} & \max\,(|x|,\ |x'|) \\ (x,x')\,{}^{\frown''}(y,y') & \widehat{=} & (x{}^\frown y,\ x'\,{}^{\frown'}y') \lhd |x| < \infty \rhd (x,\ x') \\ (x,x') \leqslant'' (y,y') & \widehat{=} & (x \leqslant y \wedge x' \leqslant y') \lhd |x| < \infty \rhd (x = y \wedge x' = y')\,. \end{array}$$

Example 2.9 The control product $(\mathit{RTime} \nearrow \{t\}) \rhd \mathit{Optimization}$ represents a real-time optimization process, which will be terminated if the timeout limit t is reached.

3 Specification of cumulative computing

Let $C = (X,\ \leqslant\ ;\ \mathbf{0},\ {}^\frown\ ;\ |\cdot|)$ be a *normal* cumulator. A specification (also called 'command') of cumulative computing is a subset $P \subseteq X$ of cumulations. The following table lists the basic imperative commands.

$\bot$	$\widehat{=}$	X	chaos (all cumulations)
$\top$	$\widehat{=}$	$\emptyset$	magic (no cumulations)
$\triangleright$	$\widehat{=}$	$\{\, x \in X \mid \lvert x\rvert < \infty \,\}$	termination (all finite cumulations)
$\triangleleft$	$\widehat{=}$	$\{\, x \in X \mid \lvert x\rvert = \infty \,\}$	nontermination (all infinite cumulations)
$\rm I\!I$	$\widehat{=}$	$\{\, \mathbf{0} \,\}$	skip (zero cumulation)
$P \mathbin{;} Q$	$\widehat{=}$	$\{\, x^\frown y \mid x \in P,\ y \in Q \,\}$	sequential composition
$P \cup Q$			nondeterministic choice (set union)
$P \cap Q$			parallel composition (set intersection)
$\sim P$	$\widehat{=}$	$X \setminus P$	negation (set complement)
$P \mid Q$	$\widehat{=}$	$(P \cap \triangleright) \cup (Q \cap \triangleleft)$	partition
ϕf	$\widehat{=}$	$\mu X \cdot f(\nu f \mid X)$	recursion (partitioned fixpoint)

Commands of cumulative computing form a Boolean complete lattice under set containment (i.e. the refinement order). The top, bottom, lub, glb and complement are $\top$, $\bot$, $\cap$, $\cup$ and $\sim$ respectively. We allow universal lub $\bigcap$ and universal $\bigcup$, which are closed under the complete lattice. Termination $\triangleright$ and nontermination $\triangleleft$ are complements of each other, i.e. $\triangleright \cup \triangleleft = \bot$ and $\triangleright \cap \triangleleft = \top$. Skip $\rm I\!I$ is the singleton set of the zero cumulation.

Sequential composition of two commands is their pointwise concatenation. Concatenation operator is associative, so is sequential composition. Skip $\rm I\!I$ is the unit of sequential composition. The (sequential) interactions between extreme commands are shown in the following table in which P is on the left and Q is at the top.

$P \mathbin{;} Q$	$\top$	$\bot$	$\triangleright$	$\triangleleft$
$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\bot$	$\triangleleft$
$\triangleright$	$\top$	$\bot$	$\triangleright$	$\triangleleft$
$\triangleleft$	$\top$	$\triangleleft$	$\triangleleft$	$\triangleleft$

Nondeterministic choice corresponds to set union. The simplest form of parallelism corresponds to set intersection and exhibits the common behaviours agreed by two commands. We will discuss more advanced forms of parallel compositions in Section 4. Nondeterministic choice and parallel composition are idempotent, commutative and distributive with each other. Sequential composition distributes over both nondeterministic choice and parallel composition. Negation is useful in specifying assumptions. It satisfies De Morgan's laws and other laws of complement of Boolean complete lattice.

The important composition, *partition* $P \mid Q$ combines the terminating cumulations of P with the nonterminating cumulations of Q. Partitions satisfy some simple but powerful laws and will be used to define recursions.

Law 1 (1) $(P \mid R) \mid Q \;=\; P \mid Q$ (2) $P \mid (R \mid Q) \;=\; P \mid Q$

(3) $P \;=\; (P \mid P) \;=\; (P \mid \top) \cup (\top \mid P)$

Law 2 (1) $P \fatsemi Q \;=\; (P|\top \fatsemi Q|\top) \mid ((\top|P \fatsemi Q) \cup (P \fatsemi \top|Q))$

(2) $P \cup Q \;=\; (P|\top \cup Q|\top) \mid (\top|P \cup \top|Q)$

(3) $P \cap Q \;=\; (P|\top \cap Q|\top) \mid (\top|P \cap \top|Q)$

(4) $\sim\! P \;=\; \sim\!(P|\top) \mid \sim\!(\top|P)$

A general recursion is normally written as an equation: $Y = f(Y)$ in which Y is called the *recursive argument*, and $f(Y)$ called the *recursion*. In this paper, the fixpoint of a recursion $f(Y)$ is denoted by $\phi Y \cdot f(Y)$ or ϕf for short. For example, if the cumulator C is $\mathit{Trace}(\mathbb{N})$, the recursion $\phi Y \cdot (\{\langle 1 \rangle\} \fatsemi Y)$ specifies a process that generates an infinite sequence of 1's.

The modeling of general recursions is subtle. Since (unbounded) nondeterminism is allowed, a recursion does not guarantee a unique fixpoint. Among all fixpoints, we must determine the fixpoint that is consistent with programmer's intuition and at the same time convenient to our semantic studies. Tarski's fixpoint theorem [17] is a standard technique to determine the least fixpoint of a *monotonic* function over a complete lattice (or a well-founded partial order if the function is known to have some fixpoint). Commands of cumulative computing can be ordered by various well-founded partial orders. The least fixpoint with regard to the refinement order $\supseteq$ is the weakest (or most chaotic) fixpoint denoted by $\mu f \;\widehat{=}\; \bigcup \{\, Y \in \mathcal{P}(X) \mid Y \supseteq f(Y) \,\}$. The least fixpoint with regard to reverse refinement order $\supseteq$ is the strongest (or most miraculous) fixpoint $\nu f \;\widehat{=}\; \bigcap \{\, Y \in \mathcal{P}(X) \mid Y \subseteq f(Y) \,\}$. We may use other orders to determine fixpoints. Various orders have been proposed in the past. All of them are applicable in some circumstances, but none of them is universally applicable.

Let $\sqsubseteq_{\divideontimes}$ be the well-founded partial order that we use. Note that $\sqsubseteq_{\divideontimes}$ should not just be a preorder, if we want to uniquely pinpoint fixpoints using Tarski's theorem. Any calculation of Tarski's least fixpoint starts from the bottom $\bot_{\divideontimes}$ of the order. The corresponding function $f(Y) = Y$ of the empty loop $\phi Y \cdot Y$ immediately reaches its least fixpoint $\bot_{\circledast}$. Since the empty loop *never* terminates, its semantics must not contain any terminating cumulation; otherwise, for example in trace semantics, if its fixpoint were chaos $\bot$, we would have an undesirable inequality $(\phi Y \cdot Y) \mathbin{;} \{\langle 1 \rangle\} \neq (\phi Y \cdot Y)$ in which the right-hand side allows the empty trace $\langle\,\rangle$ but the left-hand side does not. The inequality suggests that the behaviour of a nonterminating process could be altered if it is followed by another process that generates an event 1. Such counterintuitive interpretation is the result of the incorrect semantic assumption on the empty loop. Thus we conclude that $\bot_{\circledast} \subseteq \lhd$. On the other hand, the empty loop is an executable program that at least generates some outputs. Thus its semantics must not be empty, i.e. $\top \subset \bot_{\circledast}$. In summary, the required order $\sqsubseteq_{\circledast}$ must satisfy:

(A) $\sqsubseteq_{\circledast}$ is a well-founded partial order,
(B) $\top \subset \bot_{\circledast} \subseteq \lhd$ where $\bot_{\circledast}$ is the bottom of the order,
(C) all program compositions are $\sqsubseteq_{\circledast}$-monotonic.

Unfortunately it has been proved that such an order $\sqsubseteq_{\circledast}$ does not exist [3]. We use a new technique called *partitioned fixpoint* of which the following theorem is the basic property.

Theorem 1 *If a $\supseteq$-monotonic function f is* distributive $f(Y|\top) = f(Y|\top)|\top$ *for any command Y, then ϕf is a fixpoint such that $f(\phi f) = \phi f$.*

For example the strongest fixpoint $\nu Y \cdot (P \mathbin{;} Y)$ of the recursion $\phi Y \cdot (P \mathbin{;} Y)$ is $\top$ and hence its partitioned fixpoint can be calculated as follows:

$$\phi Y \cdot (P \mathbin{;} Y) \;=\; \mu Y \cdot (P \mathbin{;} (\top \,|\, Y)) .$$

The empty loop can be determined as a special case: $\phi Y \cdot Y = \bigcap_{\kappa}(\mathrm{II}^{\kappa} \mathbin{;} \lhd) = \lhd$ where κ is any ordinal.

Many useful specification commands can be derived from the basic ones.

P^0	$\widehat{=}$	II	zero repetition
P^n	$\widehat{=}$	$P \mathbin{;} P^{n-1}$	repetition n times $(n > 1)$
P^∞	$\widehat{=}$	$\bigcap_{n<\infty}(P^n \mathbin{;} \lhd)$	ω-infinite repetition
P^λ	$\widehat{=}$	$\bigcup_{n\in\lambda} P^n$	general repetition $(\lambda \subseteq \mathbb{N}_\infty)$
$P \rightrightarrows Q$	$\widehat{=}$	$\sim P \cup Q$	rely-guarantee specification
$\Diamond P$	$\widehat{=}$	$\rhd \mathbin{;} P \mathbin{;} \bot$	temporal operator of possibility
$\Box P$	$\widehat{=}$	$\sim\Diamond\sim P$	temporal operator of necessaity

Repetition P^n repeats the command P for n times sequentially. Zero repetition P^0 is skip. ω-Infinite repetition P^∞ represents a nonterminating loop. Note that it equals $\phi Y \cdot (P \,;\, Y)$ only when it is indeed a fixpoint that satisfies $P \,;\, P^\infty = P^\infty$. General repetition P^λ is the nondeterministic choice of n-time repetitions for all $n \in \lambda$ where λ is a subset of $\mathbb{N}_\infty$. All general repetitions are monotonic with regard to the refinement order. We use two conventions $P^* = P^{\mathbb{N}_\infty}$ and $P^\circledast = P^{\mathbb{N}}$ to denote special repetitions and assume that $P^\emptyset = \top$. The pointwise sum of two sets is defined $\lambda + \mu \mathrel{\widehat{=}} \{n + m \mid n \in \lambda,\, m \in \mu\}$, and their pointwise product is defined $\lambda \times \mu \mathrel{\widehat{=}} \{n \times m \mid n \in \lambda,\, m \in \mu\}$. Sequential composition, nondeterministic choice and parallel composition of general repetition operators can then be merged.

Law 3 (1) $P^\lambda \,;\, P^\mu = P^{(\lambda+\mu)}$ (2) $(P^\lambda)^\mu = P^{(\lambda \times \mu)}$ $(0 \notin \lambda$ *or* $\infty \notin \mu)$

(3) $P^\lambda \cup P^\mu = P^{(\lambda \cup \mu)}$ (4) $P^\lambda \cap P^\mu = P^{(\lambda \cap \mu)}$

Using partitions, we can reason about repetition's terminating and nonterminating behaviors in separate. The following laws are essential properties of repetitions.

Law 4 (1) $P^* = (P|\top)^* \,;\, (\mathrm{II}|P)$ (2) $P^\circledast = (P|\top)^\circledast \,;\, (\mathrm{II}|P)$

(3) $P^\infty = (P|\top)^\infty \cup (P|\top)^\circledast \,;\, (\top|P)$

Rely-guarantee specification is a general form of logical implication. A computation satisfies a rely-guarantee condition $P \rightrightarrows Q$ iff whenever P is satisfied Q is guaranteed. This corresponds to the rely-guarantee specifications in TLA [11] and UNITY [1] and satisfies the laws:

Law 5 (1) $P \cap (P \rightrightarrows Q) = P \cap Q$ (2) $P \rightrightarrows Q = \bot$ *iff* $P \subseteq Q$.

Temporal operator $\Diamond P$ specifies the liveness property that the computation 'eventually' behaves like the command P, while temporal operator $\Box P$ specifies the safety property that the computation 'always' behaves like the command P. Standard logical axioms of reflexivity, idempotence and seriality now become algebraic laws respectively.

Law 6 (1) $\Box P \rightrightarrows P = \bot$ (2) $\Box P \rightrightarrows \Box\Box P = \bot$

(3) $\Box P \rightrightarrows \Diamond P = \bot$

The above commands can be used to support rely-guarantee or temporal-logic style of reasoning, although the emphasis of this paper is imperative parallel programming.

4 Case study: from CSP to Timed CSP

In this section, we will try to model CSP processes as special commands of cumulative computing. The basic CSP language that we consider has the following syntax:

$$\begin{aligned} P \ = \ & \mathit{STOP} \mid \mathit{SKIP} \mid P_A \mid P \fatsemi P \mid a \rightarrow \mathit{SKIP} \mid P \setminus E \mid \\ & P \mathbin{[\!]} P \mid P \sqcap P \mid P \mathbin{|||} P \mid P \parallel P \mid \phi f\ . \end{aligned}$$

The nonterminating process STOP represents a deadlock. It generates an empty trace of events and refuses all events from its alphabet. The process SKIP always terminates successfully but also generates an empty trace of events. The command P_A denotes the *alphabetical restriction* over a process P. It requires that the process P generate only traces of events in the alphabet A. If P does not satisfy this restriction, the command P_A becomes magic that indicates the occurrence of inconsistency. $P \fatsemi Q$ is sequential composition of two processes. The process $a \rightarrow \mathit{SKIP}$ either performs a single event a before termination or waits for the event a to occur. A process $a \rightarrow P$ is equivalent to the sequential composition of $a \rightarrow \mathit{SKIP}$ and P. Event hiding $P \setminus E$ reduces the alphabet of P. Only events of P not in E are observable. $P \mathbin{[\!]} Q$ denotes external choice between two processes. Nondeterministic choice $P \sqcap Q$ becomes a disjunction. Fair-interleaving composition $P \mathbin{|||} Q$ terminates if both P and Q terminate. The trace of the composition can be any interleaving of the traces of the two processes, which must agree on the set of refused events. A parallel composition $P \parallel Q$ terminates if both processes terminate. The trace of $P \parallel Q$ in P's (or Q's) alphabet is the same as the trace of P (or Q). The composition refuses any event that is refused by either P or Q.

The failures-divergences semantics is a popular semantics of CSP [16]. To model CSP using commands of cumulative computing, we must first identify the resources involved and then represent them as appropriate cumulators.

Let Σ be the set of all events that we consider. The (finite) set of all observable events (called the *alphabet*) of a process can be formalized as a cumulator $\mathit{Alpha} \mathrel{\widehat{=}} \mathit{Fin}(\mathit{Set}(\Sigma))$ whose chop operator is set union. This allows us to expand the alphabet of a process. For example, the alphabet of the sequential composition of two processes is the union of the their alphabets. Since each process 'carries' the information about its alphabet, parallel composition no longer needs two alphabets as its parameters. This helps clarify the difference between a command (e.g. SKIP) and its alphabetical restriction (e.g. SKIP_A): the alphabet of the former is *nondeterministically arbitrary*, while that of the latter is *deterministic* on a set.

Failures-divergences model allows only finite traces, which can be represented as a restricted cumulator $\mathit{Trace} \mathrel{\widehat{=}} \mathit{Fin}(\mathit{Trace}(\Sigma))$. The trace of a CSP process is downwards-closed. Any trace's prefix is also a possible trace of the process. This reflects the assumption that any event either occurs or waits indefinitely.

The set of events refused by a waiting process (called a *refusal*) is a cumulator $\mathit{Refusal} \mathrel{\widehat{=}} \mathit{Set}'(\Sigma)$. The refusal of a process is downwards closed in the sense that any refusal's subset is also a refusal. The cumulation of trace has priority over the cumulation of refusal. This can be represented as the priority product of the two cumulators $\mathit{Trace} \ltimes \mathit{Refusal}$.

Termination becomes a resource whose cumulator is *Termin*. A nonterminating process *always* consumes infinite resource of *Termin*. If a process does not terminates (for example due to deadlock), its trace and refusal will be kept unchanged. This is achieved by a control product $\mathit{Termin} \triangleright (\mathit{Trace} \ltimes \mathit{Refusal})$.

The sequential composition of two processes may refuse the events that are refused by either process: if the first process does not terminate (for example due to deadlock), then the cumulation of trace and refusal is blocked (according to the definition of control product), and the composition refuses only the events refused by the first process; if the first process terminates and the second process has performed some new event, then the priority product forces the composition to refuse only the events refused by the second process; otherwise, if the first process terminates but the second one has started but waits indefinitely without performing its first events, their composition refuses the events refused by the second process. Note that, in CSP, since a terminating process is not ready to accept any new event until the start of a following process, its refusal is arbitrary. We also note that the refusal of the second process is downwards closed. That means, in the last case, although the priority product requires the composition to refuse the events commonly refused by both processes, these events are actually all the events that are refused by the second process.

The cumulator of CSP (denoted by the variable x) is a combination of the four cumulators of alphabet, termination, trace and refusal, which are denoted by the variables $\mathit{alf} \mathrel{\widehat{=}} (x)_1$, $\mathit{ok} \mathrel{\widehat{=}} (x)_2$, $\mathit{tr} \mathrel{\widehat{=}} (x)_3$ and $\mathit{ref} \mathrel{\widehat{=}} (x)_4$ respectively. A process can only perform events allowed by its alphabet and thus satisfy $(\mathit{tr} \upharpoonright \mathit{alf} = \mathit{tr})$. The refusal set is a subset of the alphabet: $(\mathit{ref} \subseteq \mathit{alf})$. The composite cumulator of CSP is finally obtained:

$$\mathit{CSP} \;\; \widehat{=} \;\; (\mathit{Alpha} \times (\mathit{Termin} \triangleright (\mathit{Trace} \ltimes \mathit{Refusal}))) \upharpoonright R$$

where $R \;\widehat{=}\; (\mathit{tr} \upharpoonright \mathit{alf} = \mathit{tr}) \wedge (\mathit{ref} \subseteq \mathit{alf})$. A CSP process is a set of cumulations in *CSP*. *For convenience, we may represent a set of cumulations as a predicate $P(x)$ on a variable x (whose type is a four tuple), or alternatively, a predicate $P(\mathit{alf}, \mathit{ok}, \mathit{tr}, \mathit{ref})$ on the four variables alf, ok, tr and ref.* The definitions of sequential composition and recursion of CSP are the same as those of cumulative computing. Other commands are defined in a following table (cf. [10]).

$$
\begin{array}{rcl}
\mathit{SKIP} & \widehat{=} & \mathit{ok} \wedge \mathit{tr} = \langle\,\rangle \wedge \mathit{ref} \subseteq \mathit{alf} \\
\mathit{STOP} & \widehat{=} & \neg \mathit{ok} \wedge \mathit{tr} = \langle\,\rangle \wedge \mathit{ref} \subseteq \mathit{alf} \\
a \rightarrow \mathit{SKIP} & \widehat{=} & (\mathit{tr} = \langle a \rangle \wedge \mathit{ref} \subseteq \mathit{alf}) \lhd \mathit{ok} \rhd (\mathit{tr} = \langle\,\rangle \wedge \mathit{ref} \subseteq \mathit{alf} \setminus \{a\}) \\
a \rightarrow P & \widehat{=} & (a \rightarrow \mathit{SKIP}) \fatsemi P \\
P_A & \widehat{=} & P \wedge \mathit{alf} = A \\
P \setminus E & \widehat{=} & \exists x_0 \cdot P[x_0/x] \wedge (\mathit{alf} = \mathit{alf}_0 \setminus E) \wedge (\mathit{ok} = \mathit{ok}_0) \wedge \\
 & & \quad (\mathit{tr} = \mathit{tr}_0 \upharpoonright \mathit{alf}) \wedge (\mathit{ref} = \mathit{ref}_0 \cap \mathit{alf}) \\
P \mathbin{[\,]} Q & \widehat{=} & (P \wedge Q) \lhd (\neg \mathit{ok} \wedge \mathit{tr} = \langle\,\rangle) \rhd (P \vee Q) \\
P \sqcap Q & \widehat{=} & P \vee Q \\
P \mathbin{|||} Q & \widehat{=} & \exists x_0 x_1 \cdot (P[x_0/x] \wedge Q[x_1/x] \wedge \\
 & & \quad \mathit{alf} = \mathit{alf}_0 = \mathit{alf}_1 \wedge \mathit{ok} = \mathit{ok}_0 \wedge \mathit{ok}_1 \wedge \\
 & & \quad \mathit{tr} \in (\mathit{tr}_0 \interleave \mathit{tr}_1) \wedge \mathit{ref} = \mathit{ref}_0 \cap \mathit{ref}_1) \\
P \parallel Q & \widehat{=} & \exists x_0 x_1 \cdot (P[x_0/x] \wedge Q[x_1/x] \wedge \\
 & & \quad \mathit{alf} = \mathit{alf}_0 \cup \mathit{alf}_1 \wedge \mathit{ok} = \mathit{ok}_0 \wedge \mathit{ok}_1 \wedge \\
 & & \quad \mathit{tr} \upharpoonright \mathit{alf}_0 = \mathit{tr}_0 \wedge \mathit{tr} \upharpoonright \mathit{alf}_1 = \mathit{tr}_1 \wedge \\
 & & \quad \mathit{ref} = \mathit{ref}_0 \cup \mathit{ref}_1)
\end{array}
$$

The semantic model is pleasingly simple and has some interesting differences from the failures-divergences semantics. For example, the alphabet is no longer a separate parameter but a variable whose type is a cumulator. Divergences are represented as nontermination $\lessdot$ but not chaos $\perp$. Infeasible specifications (i.e. miracles) such as $\top$ are allowed. Formal program derivation based on the refinement order is hence supported. The most significant difference lies in the modeling of recursion. Partitioned fixpoint instead of 'weakest fixpoint' is used.

A number of generalizations of CSP are now possible. For example, the original CSP allows only finite sequences and finite nondeterminism. These restrictions can be removed if we use the cumulator $\mathit{Trace}(\Sigma)$ directly. Note that the cumulator allowing infinite traces must satisfy an additional restriction $\mathit{ok} \Rightarrow (|\mathit{tr}| < \infty)$ for consistency between the cumulators of trace and termination. Using partitions, we can easily define and reason about infinite traces in the general model of cumulative computing.

Timed CSP is another non-trivial generalization of untimed CSP. Not only termination but also absolute time of termination should be represented. The cumulator *Termin* is then replaced by the cumulator of absolute time *AbsTime*.

In untimed CSP, a *failure* is a pair of trace and refusal. A process refuses to perform events in the refusal *after* performing a trace of events. In the timed failures model, a *timed failure* is a pair of timed trace and timed refusal. A process may perform a trace of events *while* refusing the events of the refusal set. To model Timed CSP, we need to replace the cumulators of trace and refusal with the cumulators of timed trace and timed refusal respectively. Let the time domain be $[0, \infty)$. A *timed trace* is a finite trace of time-stamped events with non-decreasing time points:

$$\mathit{TimedTrace} \;\widehat{=}\; \mathit{Fin}(\mathit{Trace}([0,\infty)\times\Sigma) \restriction (\forall i < |x| \cdot (x_i)_1 \leqslant (x_{i+1})_1)) \ .$$

A *refusal token* is the Cartesian product of a finite-time half-open interval and a set of events. All events from a token will be refused continuously throughout the interval of the token. A timed refusal is a finite set of refusal tokens:

$$\mathit{TimedRefusal} \;\widehat{=}\; \mathit{Fin}(\mathit{Set}(\{\, [t_1, t_2) \times A \mid 0 \leqslant t_1 < t_2 < \infty,\ A \subseteq \Sigma \,\})) \ .$$

The cumulator of Timed CSP is a combination of the cumulators of alphabet, termination, timed trace and timed refusal:

$$\mathit{TimedCSP} \;\widehat{=}\; (\mathit{Alpha} \times (\mathit{AbsTime} \triangleright (\mathit{TimedTrace} \ltimes \mathit{TimedRefusal}))) \restriction R$$

where $R \;\widehat{=}\; (\mathit{ttr} \restriction ([0,t)\times \mathit{alf}) = \mathit{ttr} \ \wedge\ \mathit{tref} \subseteq ([0,\infty)\times \mathit{alf}))$, $\mathit{alf} \;\widehat{=}\; (x)_1$ is still the alphabet, $t \;\widehat{=}\; (x)_2$ denotes the absolute time of termination, $\mathit{ttr} \;\widehat{=}\; (x)_3$ is the timed trace, and $\mathit{tref} \;\widehat{=}\; \bigcup (x)_4$ is the timed refusal. Commands of Timed CSP become cumulative specifications. For example, the command *WAIT* t_0 is a delayed form of *SKIP*. It does nothing but is ready to terminate successfully after the specified time t_0. Sequential composition and recursion are the same as those of cumulative computing. Alphabetical restriction P_A is the same as that of untimed CSP. For convenience, we use a convention $\mathit{talf} \;\widehat{=}\; [0,\infty)\times \mathit{alf}$.

$$
\begin{array}{rcl}
\mathit{SKIP} & \widehat{=} & t < \infty \wedge \mathit{ttr} = \langle\,\rangle \wedge \mathit{tref} \subseteq \mathit{talf} \\
\mathit{STOP} & \widehat{=} & t = \infty \wedge \mathit{ttr} = \langle\,\rangle \wedge \mathit{tref} \subseteq \mathit{talf} \\
\mathit{WAIT}\ t_0 & \widehat{=} & t_0 \leqslant t < \infty \wedge \mathit{ttr} = \langle\,\rangle \wedge \mathit{tref} \subseteq \mathit{talf} \\
a \rightarrow \mathit{SKIP} & \widehat{=} & (\exists t_1 < t \cdot \mathit{ttr} = \langle (t_1, a) \rangle \wedge \mathit{tref} \subseteq \mathit{talf} \setminus ([0, t_1) \times \{a\})) \\
 & & \quad \lhd (t < \infty) \rhd (\mathit{ttr} = \langle\,\rangle \wedge \mathit{tref} \subseteq \mathit{talf} \setminus ([0, \infty) \times \{a\})) \\
a \rightarrow P & \widehat{=} & (a \rightarrow \mathit{SKIP}) \mathbin{;} P \\
P \setminus E & \widehat{=} & \exists x_0 \cdot (P[x_0/x] \wedge (\mathit{alf} = \mathit{alf}_0 \setminus E) \wedge \\
 & & \quad (t = t_0) \wedge \mathit{ttr} = \mathit{ttr}_0 \upharpoonright \mathit{talf} \wedge \mathit{tref} = \mathit{tref}_0 \cap \mathit{talf}\,) \\
P \mathbin{[\!]} Q & \widehat{=} & (P \wedge Q) \lhd (t = \infty \wedge \mathit{ttr} = \langle\,\rangle) \rhd (P \vee Q) \\
P \sqcap Q & \widehat{=} & P \vee Q \\
P \mathbin{|\!|\!|} Q & \widehat{=} & \exists x_0 x_1 \cdot (P[x_0/x] \wedge Q[x_1/x] \wedge \\
 & & \quad \mathit{alf} = \mathit{alf}_0 = \mathit{alf}_1 \wedge t = \max(t_0, t_1) \wedge \\
 & & \quad \mathit{ttr} \in (\mathit{ttr}_0 \mathbin{|\!|\!|} \mathit{ttr}_1) \wedge \mathit{tref} = \mathit{tref}_0 \cap \mathit{tref}_1\,) \\
P \parallel Q & \widehat{=} & \exists x_0 x_1 \cdot (P[x_0/x] \wedge Q[x_1/x] \wedge \\
 & & \quad \mathit{alf} = \mathit{alf}_0 \cup \mathit{alf}_1 \wedge t = \max(t_0, t_1) \wedge \\
 & & \quad \mathit{ttr} \upharpoonright ([0, \infty) \times \mathit{alf}_0) = \mathit{ttr}_0 \wedge \\
 & & \quad \mathit{ttr} \upharpoonright ([0, \infty) \times \mathit{alf}_1) = \mathit{ttr}_1 \wedge \\
 & & \quad \mathit{tref} = \mathit{tref}_0 \cup \mathit{tref}_1\,)
\end{array}
$$

The empty loop whose body *may* take zero time caused a subtle problem in Timed CSP. In the original presentation of Timed-CSP semantics, the problem was circumvented by requiring any loop's body to have at least a constant delay δ. Later models do not assume the delay but cannot guarantee a valid semantics for every recursion and requires a “simple” syntactic check to avoid empty loops. This is certainly not satisfactory from a semantic point of view. Fortunately, we have solved the problem using the technique of partitioned fixpoint. The solution coincides with the traditional failures-divergences model when a loop is guarded or the body of a loop has at least a constant delay. Any recursion in Timed CSP now has a valid semantics as a cumulation command. Law 4 can be used to reason about infinite processes. For example, the recursion $\phi X \cdot (SKIP \fatsemi X)$ is the same as the command of nontermination $\lessdot$.

5 Conclusion

A cumulation is simply represented as a five tuple. The use of the volume function is aimed to simplify specifications. It transforms a potentially complex domain to the range of real numbers. In most applications, the range of real numbers is rich enough for the reasoning of limit points, continuity, density and other topological properties. We particularly focus on the common features of computational models and, more importantly, study how typical phenomena of concurrency such as reactiveness, safety and liveness, fairness, real time and branching time naturally arise from a simple semantic model that may contribute to a programming theory.

To apply the theory to a specific computational model, we need to first identify the resources involved. Each resource is then modeled as a cumulator. Finding the right cumulator is normally not difficult. Various examples have been provided in this paper. However, more experience is required to combine cumulators with appropriate constructors. Several subtly different constructors have been introduced. Their differences reflect the essential distinctions of computational models. A specification is simply a predicate on a cumulator (or a set of cumulations). If the cumulator is discrete and finite, the formalism is complete in the sense that the equality between any two specifications without recursions can be established using just the laws of predicate calculus. If recursions are present, manual calculation is required.

Denotational semantics has been blamed for not scaling up to real programming languages. For example, a domain-theoretic semantics of a language like OCCAM or BSP [18] can be over-complicated and provide little insight into reasoning. The model of cumulative computing is proposed to provide some higher-level concepts and more abstract formalism to fill the gap between low-level domain construction and real programming models.

The case study on CSP and Timed CSP posed a substantial challenge to us. The failures-divergences model includes an alphabet of events, information about termination, a trace of events and a refusal set of events. Their links with resource cumulation are not obvious. In particular, the priority-product relation between trace and refusal and is subtle, so is control product that protects trace and refusal after a deadlock. Once the composite cumulator of CSP is determined, CSP becomes a special model of cumulative computing and all laws can be inherited. More importantly, the generalization from CSP to Timed CSP simply becomes the replacement of individual cumulators. This minimizes our effort of linking and combining computational models.

Acknowledgement

This paper arose from a DPhil thesis. The author is grateful to his supervisor J.W. Sanders for various discussions, comments and suggestions. The author also gratefully acknowledge the wide-ranging comments of anonymous referees.

References

[1] Chandy, K. M. and J. Misra, "Parallel Program Design: A Foundation," Addison-Wesley, 1988.

[2] Chen, Y., "Formal Methods for Global Synchrony," Ph.D. thesis, Oxford University Computing Laboratory (2001).

[3] Chen, Y., *A fixpoint theory for non-monotonic parallelism*, in: *11th Annual Conference of the European Association for Computer Science Logic, CSL'02*, LNCS **2471** (2002), pp. 120–134, full version to appear in TCS.

[4] Chen, Y. and J. Sanders, *Logic of global synchrony*, in: *12th International Conference on Concurrency Theory*, LNCS **2154** (2001), pp. 487–501.

[5] Davies, J. and S. Schneider, *A brief history of Timed CSP*, Theoretical Computer Science **138** (1995), pp. 243–271.

[6] Dijkstra, E., *Cooperating sequential processes*, EWD 23 (1965), university of Eindhoven, The Netherlands.

[7] Francez, N., "Fairness," Springer-Verlag, 1986.

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

[9] Hoare, C. A. R., "Communicating Sequential Processes," Prentice Hall, 1985.

[10] Hoare, C. A. R. and J. He, "Unifying Theories of Programming," Prentice Hall, 1998.

[11] Lamport, L., *A temporal logic of actions*, ACM Transctions on Programming Languages and Systems **16** (1994), pp. 872–923.

[12] Mislove, M., A. Roscoe and S. Schneider, *Fixed points without completeness*, Theoretical Computer Science **138** (1995), pp. 273–314.

[13] O'Hearn, P. W., *Resource interpretations, bunched implications and the alpha-lambda-calculus*, in: J.-Y. Girard, editor, *4th Conference on Typed Lambda-Calculi and Applications*, LNCS **1581** (1999), pp. 258–279.

[14] Peterson, J. L., "Petri Net Theory and Modelling of Systems," Prentice Hall, 1981.

[15] Reed, G. and A. Roscoe, *A timed model for communicating sequential processes*, Theoretical Computer Science **58** (1988), pp. 249–261.

[16] Roscoe, A., "The Theory and Practice of Concurrency," Prentice Hall, 1998.

[17] Tarski, A., *A lattice-theoretical fixpoint theorem and its applications*, Pacific Journal of Mathematics **5** (1955), pp. 285–309.

[18] Valiant, L., *A bridging model for parallel computation*, Communications of the ACM **33** (1990), pp. 103–111.