Layering Techniques for Development of Parallel Systems

An Algebraic Approach *

Mannes Poel & Job Zwiers,
University of Twente,
Dept. of Computer Science
P.O. Box 217,
7500 AE Enschede,
The Netherlands
E-mail:{mpoel,zwiers}@cs.utwente.nl

Abstract

A process language is presented which makes a clear distinction between temporal order and causal order. This allows for several algebraic laws that are particularly interesting for the *design* of concurrent systems. One of these is an algebraic formulation of the communication closed layers principle by [EF82]. These laws suffice to rewrite process terms that avoid specification of temporal ordering into a unique normal form. Other transformations allow for *gradually* imposing temporal ordering on an already functionally correct design. The combination of such laws enables a design strategy where architecture independent designs are transformed towards a form that matches a particular implementation architecture. We apply this style of design to various distributed algorithms, including an algorithm for the "point-in-polygon" problem transformed to a form suitable for pipelined execution on a tree network, and the Floyd-Warshall algorithm for the all-points shortest path transformed to a form suitable for execution on a SIMD architecture.

1 Introduction

In order to have a transformational algebraic approach that suits both specification and design of concurrent systems, a clear distinction should be made between temporal order and causal order between actions. In this paper we present a process language where a distinction is made between language constructs for specifying temporal order, such as the sequential composition operator, and causal order, using the *layer composition* operator, cf. [JPZ91]. Layer composition, denoted by $P \bullet Q$, gives rise to several important algebraic laws. For instance the communication closed layers principle from [EF82] can be formulated as the following algebraic law:

$$(P \bullet Q) \parallel (R \bullet S) = (P \parallel R) \bullet (Q \parallel S),$$

provided there is no "conflict" (communication) between P and S, and between Q and R. Another algebraic law, called the Left-Right Movers law states in its simplest form that

$$(P \bullet Q) \bullet (R \bullet S) = (P \bullet R) \bullet (Q \bullet S)$$

if there is no conflict between Q and R. The above laws would not be valid if we replaced layer composition by sequential composition.

Layer composition is a valuable tool in the *initial design stage* of a system. Such operations allow for an architecture independent design strategy. In this initial phase of design, the above laws together

*Part of this work has been supported by Esprit/BRA Project 6021 (REACT)

with an *expansion* theorem make it possible to rewrite a process term that avoids specification of temporal order into an unique normal form. This normal form is the maximal parallelization of that process term, and resembles the normal form for Mazurkiewicz traces. Other transformations and implementation relations, cf. section 2, then allow for a transformation of this normal form towards a process that can be implemented on a particular architecture.

This style of design is demonstrated on various distributed algorithms. First the "point-in-polygon" algorithm presented in the book of Akl, [Akl89] is considered. After removing all the irrelevant temporal order, but preserving the causal order induced by the temporal order (i.e. changing every sequential composition in a layer composition), this algorithm is rewritten into normal form. Afterwards this normal form is transformed to several distributed processes, all suitable for pipelining on a tree network.
Next the Floyd-Warshall algorithm for the "all-points shortest path" problem is discussed. Again after all the temporal order is deleted, but the induced causal order is preserved, this algorithm is transformed into normal form. Afterwards several optimal parallelizations are obtained, both for a **CREW-PRAM** and an **EREW-PRAM**.
Other applications to for instance protocols can be found in [JZ92b, JZ92a], and to parsing in [JPSZ91].

In section 2 we introduce the language together with informal discussion of the semantics. Also the algebraic laws and the theorem on normal forms are presented. In section 3 we apply the transformational algebraic approach to the point in polygon algorithm on a tree network. Section 4 gives several transformations of the Floyd-Warshall algorithm for the all-points shortest path problem. One transformation is suitable for a **CREW-PRAM** and another for an **EREW-PRAM**. Both are optimal parallelizations of the sequential Floyd-Warshall algorithm.

2 Language, Algebraic Laws, and Normal Forms

In this section we introduce a language which contains both layer composition and sequential composition. Assume that processes perform actions a that read and write shared variables $x, y, z, \ldots \in Var$, and perform (boolean valued) tests on shared variables. We employ the usual (simultaneous) assignment notation $\mathrm{x} := \mathrm{f}$ where x and f are a list of variables and a list of expressions. Such assignments are guarded by means of a boolean expression b which must evaluate to true before executing the assignment. For such actions, that we denote by $b \,\&\, \mathrm{x} := \mathrm{f}$, the evaluation of the guard together with the assignment constitute a single *atomic* action. When the guard b is identically true, we omit it and employ the usual simultaneous assignment notation $\mathrm{x} := \mathrm{f}$. Similarly we regard boolean tests b as degenerate cases where the assignment part has been left out. Such guards can be used to model more conventional constructs. For instance we use

if b **then** P **else** Q **fi** as an abbreviation for $(b \bullet P)$ or $(\neg b \bullet Q)$

For an action a of the form $b \,\&\, \mathrm{x} := \mathrm{f}$ the set of variables $\{\mathrm{x}\}$ is called the write-set $W(a)$ of a. Similarly, we define the read-set $R(a)$ as the set of variables occurring (free) in b and the expression list f. Finally we define the *base* of a as $base(a) = R(a) \cup W(a)$. Two actions a_0 and a_1 *conflict* if one of them writes a variable x that is read or written by the other one. Formally we define a conflict relation on actions, denoted by $a_0 — a_1$ as follows:

$a_0 — a_1$ iff $W(a_1) \cap base(a_0) \neq \emptyset$ or $W(a_0) \cap base(a_1) \neq \emptyset$

Other models can be easily obtained by changing the conflict relation, for example by introducing also read-read conflicts, which is in correspondence with the **EREW-PRAM**.
The syntax for DL is as follows:

$P \in DL$,

$P ::= b \,\&\, \mathrm{x} := \mathrm{f} \mid P \parallel Q \mid P \bullet Q \mid P ; Q \mid P \text{ or } Q$

$\mid$ **skip** $\mid$ **empty** $\mid P \backslash x \mid \langle P \rangle \mid$ **io**(P)

We will now provide the intuition for the language operations of DL. A process P as a whole denotes the set $[\![P]\!]$ of all possible runs for that system. Execution of an action a results in a single

event. Therefore, actions are executed atomically. Our semantic domain is such that events, say e_0 and e_1, are in conflict, then they are ordered. Hence each run consists of a set of events, an a partial order $\longrightarrow$ on events, such that events which are in conflict are ordered with respect to $\longrightarrow$. Summarizing:
Each run $(E, \longrightarrow)$ of P satisfies:

- $(E, \longrightarrow)$ is a pomset of events, i.e. a partially ordered multiset.
- $\longrightarrow$ is a partial order on E such that if e and e' are in *conflict* then e and e' are ordered with respect to $\longrightarrow$.

Consequently we regard two processes P and Q as equal, denoted by $P = Q$ iff their sets of possible runs are equal, i.e. iff $[\![P]\!] = [\![Q]\!]$.

For *parallel composition*, the order that necessarily must exists between conflicting P and Q events is *nondeterministically* determined. The nondeterministic choices for different pairs of conflicting events are of course subject to the condition that the order must remain a partial order, so certain choices are excluded. Each choice corresponds to a (potential) different net effect of the whole run.

For *layer composition* $P \bullet Q$ the situation is the reverse: any P event e_0 precedes any Q event e_1 with which it conflicts. This resembles the sequential composition construct, but there is a substantial difference. For sequential composition $P\,;\,Q$ *all* P events precede *any* Q event . This is not so for layer composition. In the latter case any Q event e must wait only for its causal predecessors, implying that it need not wait for all P events.

Nondeterministic choice P or Q is a straightforward construct that either executes P or Q.

The process **skip** performs no action at all, and the **empty** process, cannot perform any computation at all, not even the computation executed by **skip** which contains no events. Both processes aid in formulating some algebraic properties of DL, where they act as a unit element and as a zero element respectively.

Atomic execution. Atomic brackets $\langle P \rangle$ serve to indicate that P should execute "atomically", i.e. without interference by other processes.

The *hiding* construct $P\backslash x$ hides the variable x in each run of P,i.e. it is removed from the write-set and read-set of each event. (Moreover events with empty read- and write-set are removed from the run.) The complement of the hiding operator is the projection operator; let S be a set of variables and let A be all the variables of a process P then

$$P_{|S} \stackrel{\text{def}}{=} P\backslash(A - S)$$

All the variables except those contained in S are hidden in each run of P.

$\mathbf{io}(P)$ denotes execution of a single action that captures the net effect of executing P without admitting interference by other events. The $\mathbf{io}(\cdot)$ operation is also called the *contraction* operation, since it contracts complete P runs into single events. Intuitively $\mathbf{io}(P)$ represents the input-output behavior of a process P if we execute that process in isolation, i.e. without interference from outside. This operation induces an interesting process equivalence, called IO-equivalence, and an associated *implementation* relation, denoted by $P \overset{IO}{sat} Q$.

$$P \stackrel{IO}{=} Q \text{ iff } \mathbf{io}(P) = \mathbf{io}(Q), \text{ and } P \overset{IO}{sat} Q \text{ iff } \mathbf{io}(P) \subseteq \mathbf{io}(Q).$$

Such equivalences play an important role in the book by K.R. Apt and E.-R. Olderog [AO]. Specification of what is often called the *functional behavior* of a process P is really a specification of $\mathbf{io}(P)$, i.e. of the IO-equivalence class of P. The $\mathbf{io}(\cdot)$ operation does (obviously) not distribute through parallel composition. For the case of layer composition though, we have the following laws:

$$P \bullet Q \stackrel{IO}{=} \mathbf{io}(P) \bullet \mathbf{io}(Q) \text{ and } P\,;\,Q \stackrel{IO}{=} P \bullet Q$$

The intuition here is that although execution of "layer" P might overlap execution of "layer" Q *temporally*, one can pretend that all of P, here represented as an atomic action $\mathbf{io}(P)$, precedes all of Q as far as IO behavior is concerned.

2.1 Algebraic Laws

In this section we provide some algebraic laws for model informally introduced here and extensively studied in [JPZ91]. The well-known laws for sequential composition, such as associativity, are not stated.

Lemma 2.1
Commutativity and Associativity:

$$\begin{array}{rcll} P \parallel Q & = & Q \parallel P & \text{(COM1)} \\ P \text{ or } Q & = & Q \text{ or } P & \text{(COM2)} \\ P \parallel (Q \parallel R) & = & (P \parallel Q) \parallel R & \text{(ASSOC1)} \\ P \bullet (Q \bullet R) & = & (P \bullet Q) \bullet R & \text{(ASSOC2)} \\ P \text{ or } (Q \text{ or } R) & = & (P \text{ or } Q) \text{ or } R & \text{(ASSOC3)} \end{array}$$

Distributivity:

$$\begin{array}{rcll} P \parallel (Q \text{ or } R) & = & (P \parallel Q) \text{ or } (P \parallel R) & \text{(DIST1)} \\ P \bullet (Q \text{ or } R) & = & (P \bullet Q) \text{ or } (P \bullet R) & \text{(DIST2)} \\ (P \text{ or } Q) \bullet R & = & (P \bullet R) \text{ or } (Q \bullet R) & \text{(DIST3)} \end{array}$$

Idempotency:

$$P \text{ or } P \;=\; P$$

Units and zeros:

$$\begin{array}{lcccll} \textbf{skip} \parallel P & = & P \parallel \textbf{skip} & = & P & \text{(SKIP1)} \\ \textbf{skip} \bullet P & = & P \bullet \textbf{skip} & = & P & \text{(SKIP2)} \\ \textbf{empty or } P & = & P \textbf{ or empty} & = & P & \text{(EMPTY1)} \\ \textbf{empty} \parallel P & = & P \parallel \textbf{empty} & = & \textbf{empty} & \text{(EMPTY2)} \\ \textbf{empty} \bullet P & = & P \bullet \textbf{empty} & = & \textbf{empty} & \text{(EMPTY3)} \end{array}$$

□

More interesting is the relationship between parallel composition and layer composition. We can formulate here a (simple form of) the principle of communication closed layers in the form of an algebraic law. The communication closed layers law (CCL) deviates somewhat from the usual style of algebraic laws in that there is a (syntactic) side condition that should be checked concerning conflicts between processes. Let $act(P)$ denote the (finite) set of actions that (syntactically) occur in a DL process P. Let $\mathcal{A}ct$ denote the set of actions, we extend the conflict relation on $\mathcal{A}ct$ to sets of $\mathcal{A}ct$ elements as follows: for $X, Y \subseteq \mathcal{A}ct$,

$X — Y$ iff there exist $a \in X$, $b \in Y$ such that $a — b$.

Conflicts between DL processes are then defined thus: $P — Q$ iff $act(P) — act(Q)$.

As usual, $P \not— Q$ denotes that $P — Q$ is not the case, i.e. P actions do not conflict with Q actions.

The CCL laws can now be formulated as follows.

Lemma 2.2
Communication Closed Layers:

Provided that $P \not— S$, and $Q \not— R$:

$$\begin{array}{rcll} (P \bullet Q) \parallel (R \bullet S) & = & (P \parallel R) \bullet (Q \parallel S) & \text{(CCL)} \\ (P \bullet Q) \parallel S & = & P \bullet (Q \parallel S) & \text{(CCL-L)} \\ (P \bullet Q) \parallel R & = & (P \parallel R) \bullet Q & \text{(CCL-R)} \\ Q \parallel R & = & Q \bullet R & \text{(Independence)} \end{array}$$

□

In order to state a generalized version of the CCL and related laws, we introduce the abbreviations:

for $i \leftarrow [n \ldots m]$ **dopar** $P(i)$ **rof** abbreviating: $P(n) \parallel \cdots \parallel P(m)$

for $i \leftarrow [n \ldots m]$ **layer** $P(i)$ **rof** abbreviating: $P(n) \bullet \cdots \bullet P(m)$

for $i \leftarrow [n \ldots m]$ **choice** $P(i)$ **rof** abbreviating: $P(n)$ **or** $\cdots$ **or** $P(m)$

for $i \leftarrow [n \ldots m]$ **doseq** $P(i)$ **rof** abbreviating: $P(n) ; \cdots ; P(m)$

Lemma 2.3

- *Generalized Communication Closed Layers Law.*
  Assume that if there is a conflict between P_{ij} and P_{kl} then either $i = k$ or $j = l$ is satisfied, for $1 \leq i, j, k, l \leq n$. Then
  for $i \leftarrow [1 \ldots n]$ **dopar**
  for $j \leftarrow [1 \ldots m]$ **layer** $P_{i,j}$ **rof**
  rof
  =
  for $j \leftarrow [1 \ldots m]$ **layer**
  for $i \leftarrow [1 \ldots n]$ **dopar** $P_{i,j}$ **rof**
  rof
- *Left-Right Movers Law.*
  Assume that, for $1 \leq i < k \leq n$ and $1 \leq l < j \leq m$, $P_{i,j} \not\rightarrow P_{k,l}$. Then
  for $i \leftarrow [1 \ldots n]$ **layer**
  for $j \leftarrow [1 \ldots m]$ **layer** $P_{i,j}$ **rof**
  rof
  =
  for $j \leftarrow [1 \ldots m]$ **layer**
  for $i \leftarrow [1 \ldots n]$ **layer** $P_{i,j}$ **rof**
  rof

□

2.2 Normal Forms

Let DL' denote the language DL with the sequential composition operator omitted. If P is in DL' then for each P run $(E, \longrightarrow)$ the order $\longrightarrow$ is generated by conflicts only. That is, if $e \in E$ and e' is a direct successor of e with respect to $\longrightarrow$, then e and e' are in conflict. (This is not the case if one adds the sequential composition to the language.)

Define layer $L(k)$ as

$$L(k) = \{e \in E \mid \text{the longest chain below } e \text{ has length } k\}$$

Since runs are assumed to be finite, $L(k) = \emptyset$ for sufficient large k. Enumerate the elements in each layer $L(k)$ such that

$$L(k) = \{e_{k,j} \mid 1 \leq j \leq l(k)\}$$

where $l(k)$ is the number of elements in $L(k)$.
Observe that there can be no conflict between events in the same layer, and for all events e in layer $L(k)$, $k > 1$, there exists an event e' in layer $L(k-1)$ such that $e \text{---} e'$. The run $(E, \longrightarrow)$ is can be denoted syntactically by:

for $k \leftarrow [0 \ldots n]$ **layer** **for** $j \leftarrow [1 \ldots l(k)]$ **dopar** $e_{k,j}$ **rof** **rof**

where n is such that

$$(0 \leq k \leq n \;\Rightarrow\; L(k) \neq \emptyset) \;\wedge\; k > n \;\Rightarrow\; L(k) = \emptyset$$

(Formally speaking $e_{k,j}$ is not an action but an event, and should replace $e_{k,j}$ by $\mu(e_{k,j})$, where $\mu(e_{k,j})$ is the action corresponding to the event $e_{k,j}$.) This can be done for every run of P, hence if we let r denote the number of runs of P then

P = **for** $i \leftarrow [1 \ldots r]$ **choice**
for $k \leftarrow [1 \ldots n(r)]$ **layer** **for** $j \leftarrow [1 \ldots k(n(r))]$ **dopar** $e_{i,k,j}$ **rof** **rof**
rof

The above decomposition of P has the following properties:

- For each j and j', with $j \neq j'$, $e_{i,k,j} \not\rightarrow e_{i,k,j'}$.

- For each $k > 1$ and each $e_{i,k,j}$ there exists an event $e_{i,k-1,j'}$ such that $e_{i,k,j} \frown e_{i,k-1,j'}$.

Hence we have written P as a choice over layered maximal parallel processes. This above decomposition of P leads to following definition

Definition 2.4 *Normal Form*
A normal form is a process term of the form

for $i \leftarrow [1 \ldots r]$ **choice**
  for $k \leftarrow [1 \ldots n(r)]$ **layer**
    for $j \leftarrow [1 \ldots k(n(r))]$ **dopar** $e_{i,k,j}$ **rof**
  rof
rof

where each $e_{i,k,j}$ is an elementary process term. Moreover the process term should satisfy

1. For each j and j', with $j \neq j'$, $e_{i,k,j} \not\frown e_{i,k,j'}$.
2. For each $k > 1$ and each $e_{i,k,j}$ there exists an event $e_{i,k-1,j'}$ such that $e_{i,k,j} \frown e_{i,k-1,j'}$.
3. All mutually distinct branches of the **choice** construct are syntactically different.

□

This normal form resembles the normal form for Mazurkiewicz traces.

It follows from the observations above that each process term in DL' has a normal form. But there is more to it:

Theorem 2.5 *Expansion theorem*
Let $L_P \bullet P$ and $L_Q \bullet Q$ be process terms in DL' such that any two actions in L_P and any two actions in L_Q are non-conflicting. Furthermore assume that for every action $a'_P \in P$ ($a'_Q \in Q$) there exists an action $a_P \in L_P$ ($a_Q \in L_Q$) such that $a'_P \stackrel{*}{\frown} a_P$ ($a'_Q \stackrel{*}{\frown} a_Q$), where $\stackrel{*}{\frown}$ is the transitive closure of the conflict relation $\frown$. Then

$(L_P \bullet P) \parallel (L_Q \bullet Q) =$
  for $a_P \in L_P$ **choice** $a_P \bullet (((L_P - \{a_P\}) \bullet P) \parallel (L_Q \bullet Q))$ **rof**
or
  for $a_Q \in L_Q$ **choice** $a_Q \bullet ((L_P \bullet P) \parallel ((L_Q - \{a_Q\}) \bullet Q))$ **rof**

□

The expansion theorem, together with the Left-Right Movers law, is crucial for transforming every process term P in DL' into normal form. The normal form is the maximal parallelization of P.

Theorem 2.6
Each process term $P \in DL'$ has an unique normal form (up to permutations of the indices). Moreover P can be algebraically transformed into this normal form in an algorithmic way using the algebraic laws of section 2.1 and the expansion theorem, theorem 2.5. □

A proof can be found in [PZ92].

3 The 'Point in polygon' algorithm: Pipelining on a tree network

The aim of this section is to show how to transform an *initial, algorithmic design* to a form that is suitable for implementation on a *pipelined architecture*. The (functional) correctness of the initial design is not our concern; we assume that it is the result from an initial design stage where it has been developed from a specification of the required *functional* behavior. We concentrate on the stage *following* the initial design phase, where not only functional correctness is of importance, but where also the architecture of the implementation must be taken into account.

The functional specification of the algorithm, that we do not formalize here, amounts to the following: Given a (fixed) polygon with edges $E_1, \cdots, E_N$ and a set $\{p_0, p_1, \cdots, p_m\}$ of points in a two dimensional Euclidean space, it is required to determine which of the points in $\{p_0, p_2, \ldots, p_m\}$ lay inside the polygon.

Our initial design has been essentially taken over from [Akl89]. It is based on the fact that a point p lies inside the polygon if and only if the vertical line through p intersects an odd number of edges above p. Thus the problem reduces to computing the number of intersections above p. The intuition for the algorithm in [Akl89] is as follows. Assuming that $N+1$ is a power of 2, put $s = \log(N+1)$. A tree-like arrangement of processes of depth s is used, containing one process for each edge. The coordinates of the points are stored in an array c, with $c[a]$ the coordinates of point p_a, $a = 0, \ldots, m$. The coordinates of the candidate points p are read by the top process and are broadcast during so called descend phases. A separate descend phase is executed for each 'level' in the tree, starting with the top node and ending with the level consisting of all leaf nodes. Each process receives the coordinates of the candidate point, determines locally whether the vertical line through p intersects the edge associated with that process, and broadcasts the results "downwards" to its children. After these descend phases, the total number of intersections is calculated during a number of ascend phases, where processes add together partial counts calculated by their children. If this count reaches the top process, then this top process assigns the appropriate boolean value to $inside[a]$, where *inside* is a boolean array, such that $inside[a]$ holds if and only if the point p_a lies in the polygon. The algorithm in [Akl89] is presented in the form of a *sequential composition of "layers"* $D(l)$ *and* $A(l)$, each consisting of independent, parallel executed, actions. Communication between processes is by means of shared variables, where variables d_i and p_i are used during the descend phases and variables u_i during ascend phases. (Apart from these, there are *local* variables s_i, t_i, and q_i. q_i is used to store a local copy of coordinates of the point under consideration) Each descend layer $D(l)$ is itself divided into a reading phase $D^R(l)$ and a writing phase $D^W(l)$,and similarly, $A(l)$ is split into $A^R(l)$ and $A^W(l)$. The layer corresponding to the "leaf" nodes is an exception; there is no writing for the descend phase, and no reading for the ascend phase. The algorithm given in Akl [Akl89], adapted to our notation, is:

Program 0

for $a \leftarrow [0 \ldots m]$ **doseq**

$d_1 := 0$; $p_1 := c[a]$;

for $l \leftarrow [1 \ldots s-1]$ **doseq** $D^R(l)$; $D^W(l)$ **rof** ;

$D^R(s)$; $A^W(s)$;

for $l \leftarrow [s-1 \ldots 1]$ **doseq** $A^R(l)$; $A^W(l)$ **rof** ;

$inside[a] := odd(u_1)$

rof

This describes the "layered" structure of the algorithm. The layers $D^R(l)$, $D^W(l)$, $A^R(l)$ and $A^W(l)$ can each be described by means of parallel composition of independent actions. This implies that there is no interference among parallel processes, and consequently the (functional) correctness of the algorithm can be shown relying essentially on techniques for *sequential* programs, as explained for instance in [AO91]. The reason for this is that for independent processes P and Q, parallel composition $P \parallel Q$ is (semantically) identical to layer composition $P \bullet Q$; the latter in turn is, though not identical, IO-equivalent to sequential composition $P\,;\,Q$.

The layers for the algorithm presented in [Akl89] can be specified as follows
The descending read phase for level l:

$D^R(l) =$ **for** $j \leftarrow [2^{l-1} \ldots 2^l - 1]$ **dopar**
if $Intersects(e_j, p_j)$ **then** $s_j := d_j + 1$ **else** $s_j := d_j$ **fi** $\parallel$ $q_j := p_j$
rof

The descending write phase:

$D^W(l) =$

for $j \leftarrow [2^{l-1} \ldots 2^l - 1]$ dopar $p_{2j} := q_j \parallel p_{2j+1} := q_j \parallel d_{2j} := s_j \parallel d_{2j+1} := 0$ rof

The ascending read phase:

$A^R(l) =$ for $j \leftarrow [2^{l-1} \ldots 2^l - 1]$ dopar $t_j := u_{2j} + u_{2j+1}$ rof

Finally the ascending write phase is given by:

$A^W(l) =$ for $j \leftarrow [2^{l-1} \ldots 2^l - 1]$ dopar $u_j := t_j$ rof

except for $l = s$, in that case:

$A^W(s) =$ for $j \leftarrow [(N+1)/2 \ldots N]$ dopar $u_j := s_j$ rof

The presentation in [Akl89] is in terms of *sequential* composition of layers. From the point of view of functional correctness this is unnecessary; replacing all sequential composition by *layer composition* results in a process that is IO-equivalent. Moreover, this allows for algebraic manipulation that would be invalid for the version based on sequential composition. In particular, the layer composition version allows for *overlapping* execution of different layers, resulting in a *pipelined* execution where layers that have to be executed for different candidate points are executed in parallel. Thus, we will use, as starting point for a series of transformations, the following version:

Program 1

for $a \leftarrow [0 \ldots m]$ layer

$G(a) \bullet$

for $l \leftarrow [1 \ldots s-1]$ layer $D^R(l) \bullet D^W(l)$ rof $\bullet$

$D^R(s) \bullet A^W(s) \bullet$

for $l \leftarrow [s-1 \ldots 1]$ layer $A^R(l) \bullet A^W(l)$ rof $\bullet$

$P(a)$

rof

where

$G(a) = d_1 := 0 \parallel p_1 := c[a]$ and $P(a) = inside[a] := odd(u_1)$

First we will rename the processes $D^R(l)$, $D^W(l)$, $A^R(l)$ and $A^W(l)$, according to read or write actions. Put $t = 2s$ and define

$$R(l) = \begin{cases} D^R(l) & \text{if } 1 \leq l \leq s \\ A^R(t-l) & \text{if } s+1 \leq l \leq t-1 \end{cases}$$

$$W(l) = \begin{cases} D^W(l) & \text{if } 1 \leq l \leq s-1 \\ A^W(t-l) & \text{if } s \leq l \leq t-1 \end{cases}$$

Then Program 1 can be rewritten as

Program 1'

for $a \leftarrow [0 \ldots m]$ layer

$G(a) \bullet$

for $l \leftarrow [1 \ldots t-1]$ layer $R(l) \bullet W(l)$ rof $\bullet$

$P(a)$

rof

Conceptually we chanced the tree structure in a reflected tree structure, two identical trees with the leaves merged, such that all the data flows from top to bottom.

Observe that the above program is actually in pseudo normal form. It can be transformed into normal form, using the Left-Right Movers law and the Independence law, where one should check the side-conditions for these laws based on the following pattern of conflicts:

$R(j)$—$R(j)$, $W(j)$—$W(j)$, $R(j)$—$W(j)$ and $R(j)$—$W(j-1)$

Indeed, using that **skip** is a unit for layer composition, the above program is (semantically) equal to:

for a←$[0 \ldots m]$ **layer** **for** l←$[0 \ldots m+t]$ **layer** $\tilde{R}(a,l) \bullet \tilde{W}(a,l)$ **rof** **rof**

where

$$\tilde{R}(a,l) = \begin{cases} P(a) & \text{if } l = a+t \\ R(l-a) & \text{if } a < l < a+t \\ \textbf{skip} & \text{otherwise} \end{cases}$$

$$\tilde{W}(a,l) = \begin{cases} G(a) & \text{if } l = a \\ W(l-a) & \text{if } a < l < a+t \\ \textbf{skip} & \text{otherwise} \end{cases}$$

The side-conditions for the Left-Right Movers Law are fulfilled, hence the program can be transformed into

for l←$[0 \ldots m+t]$ **layer** **for** a←$[0 \ldots m]$ **layer** $\tilde{R}(a,l) \bullet \tilde{W}(a,l)$ **rof** **rof**

Again applying the Left-Right Movers and Independence Law to the inner layered loop gives that the program equals

Program 2

for l←$[0 \ldots m+t]$ **layer**

for a←$[0 \ldots m]$ **dopar** $\tilde{R}(a,l)$ **rof** $\bullet$

for a←$[0 \ldots m]$ **dopar** $\tilde{W}(a,l)$ **rof**

rof

This last transformation results, for large enough m and again using that **skip** is unit for layer composition, in a program consisting of a phase where the tree is gradually filled, a phase where all the nodes of the tree execute simultaneously, and a final phase where the tree is emptied. For instance the layered loop

for a←$[0 \ldots m]$ **dopar** $\tilde{R}(a,l)$ **rof**

equals, by definition of $\tilde{R}(a,l)$

if $(l-t) \geq 0$ **then** $P(l-t)$ **fi** $\parallel$ **for** a←$[max(0, l-t+1) \ldots min(m, l-1)]$ **dopar** $R(l-a)$ **rof**

which on it's turn equals, by substituting $i = l-a$

if $(l-t) \geq 0$ **then** $P(l-t)$ **fi** $\parallel$ **for** i←$[max(1, l-m) \ldots min(t-1, l)]$ **dopar** $R(i)$ **rof**

If we apply similar substitutions to the layered loop with $\tilde{W}(a,l)$, the resulting program is

Program 3

for l←$[0 \ldots m+t]$ **layer**

(**if** $(l-t) \geq 0$ **then** $P(l-t)$ **fi** $\parallel$

for i←$[max(1, l-m) \ldots min(t-1, l)]$ **dopar** $R(i)$ **rof**) $\bullet$

(**if** $l \leq m$ **then** $G(l)$ **fi** $\parallel$

for i←$[max(1, l-m) \ldots min(t-1, l)]$ **dopar** $W(i)$ **rof**)

rof

The variable boundaries at the inner parallel loops are due to "filling" of the tree, for $0 \leq l \leq t-1$, and "emptying" of the tree, for $m+1 \leq l \leq m+t$. The "filling" of the tree consists of two phases

analogous to the descend and ascend phases in the original algorithm. First the information is broadcast "downwards", layer after layer, into the tree, until this downwards fill reaches the leaves. Then the results are broadcast "upwards", again layer after layer, towards the root of the tree, until it reaches layer 1. At that moment the tree is filled, this is the case for $l = t - 1$. Thereafter all cells compute in parallel for $t \leq l \leq m$ and afterwards the tree is emptied in the reverse order for $m < l \leq m + t$.

Although this program admits pipelined execution, it has the disadvantage that at each moment of time there is a different set of active processes. This is well known for pipelining in general: there is a phase where the pipeline is gradually 'filled', a phase where all processes in the pipeline are simultaneously executing, and a final phase where the pipeline is 'emptied'. This picture of "filling" and "emptying" is adequate on the abstraction level of *processes* only. On a low level where allocation of processes to *processors* is considered, there is no corresponding "starting" and "halting" of processors. Rather, during the filling and emptying phase there will be processors executing the same algorithm as others, but on non-relevant data so to say. We model this by adding 'dummy' actions to Program 3, in order to obtain a regular pattern.

Program 4

for $l \leftarrow [0 \ldots m + t]$ **layer**

(**if** $(l - t) \geq 0$ **then** $P(l - t)$ **fi** $\parallel$

for $j \leftarrow [1 \ldots t - 1]$ **dopar** $R(j)$ **rof**) $\bullet$

(**if** $l \leq m$ **then** $G(l)$ **fi** $\parallel$

for $j \leftarrow [1 \ldots t - 1]$ **dopar** $W(j)$ **rof**)

rof

Although this program doesn't semantically equal program 3, it has the property that it preserves functional correctness. More precisely program 4 projected onto the variables c and ***inside*** is semantically equal to program 3 projected onto the variables c and ***inside***. Hence program 4 has the IO behavior with respect to c and ***inside*** as the original program 1. This can be seen by applying the inverse of the transformation steps above to program 4.

Next we take a step in the design that no longer preserves semantic equality, but only IO-equivalence, by imposing *extra* order by replacing the layer composition by a corresponding sequential composition in program 4. This cannot affect functional correctness, but it does allow for allocation of processes on (sequentially executing) processors.

Program 6

for $l \leftarrow [0 \ldots m + t]$ **doseq**

(**if** $(l - t) \geq 0$ **then** $P(l - t)$ **fi** $\parallel$ **for** $j \leftarrow [1 \ldots t - 1]$ **dopar** $R(j)$ **rof**) ;

(**if** $l \leq m$ **then** $G(l)$ **fi** $\parallel$ **for** $j \leftarrow [1 \ldots t - 1]$ **dopar** $W(j)$ **rof**)

Note that for each sequential phase, there are at most $(3N + 1)/2$ and at least $(3N - 1)/2$ processes active, which suggest a rather obvious allocation onto $(3N + 1)/2$ processors. In essence there are $(3N + 1)/2$ processes executing in parallel their contribution to some read phase and afterwards there are $(3N + 1)/2$ processes executing part of some ascend phase. Note that the *structure* of the program matches the class of SIMD machines, where a number of processors execute in lockstep the same (parameterized) program.

Applying the Independence law and commutativity of parallel composition to the program fragment

(**if** $(l - t) \geq 0$ **then** $P(l - t)$ **fi** $\parallel$ **for** $j \leftarrow [1 \ldots t - 1]$ **dopar** $R(j)$ **rof**) $\bullet$

(**if** $l \leq m$ **then** $G(l)$ **fi** $\parallel$ **for** $j \leftarrow [1 \ldots t - 1]$ **dopar** $W(j)$ **rof**)

yields the following program fragment

for $j \leftarrow [1 \ldots s]$ dopar $R(j)$ rof •

(for $j \leftarrow [1 \ldots s-1]$ dopar $W(j)$ rof ‖ if $l \leq m$ then $G(l)$ fi) •

(for $j \leftarrow [s+1 \ldots t-1]$ dopar $R(j)$ rof ‖ if $(l-t) \geq 0$ then $P(l-t)$ fi) •

for $j \leftarrow [s \ldots t-1]$ dopar $W(j)$ rof

Invoking the definition of $R(j)$ and $W(j)$ and replacing layer composition by sequential composition results in $N+1$ processes that alternate between four phases, corresponding to the read/write and the descend/ascend phase. As before, the algorithm matches an SIMD architecture. The final results is

Program 7

for $l \leftarrow [0 \ldots m+t]$ doseq

for $j \leftarrow [1 \ldots s]$ dopar $D^R(j)$ rof ;

(for $j \leftarrow [1 \ldots s-1]$ dopar $D^W(j)$ rof ‖ if $l \leq m$ then $G(l)$ fi) ;

(for $j \leftarrow [1 \ldots s-1]$ dopar $A^R(j)$ rof ‖ if $(l-t) \geq 0$ then $P(l-t)$ fi) ;

for $j \leftarrow [1 \ldots s]$ dopar $A^W(j)$ rof

rof

In this program each cell, *Cell*(i), $1 \leq i \leq N$, executes the following program:

Cell(i) =

for $l \leftarrow [0 \ldots m+t]$ doseq

$q_i := p_i$;

if $Intersects(e_i, q_i)$ then $s_i := d_i + 1$ else $s_i := d_i$ fi ;

if $i < 2^{s-1}$ then $p_{2i} := q_i$; $p_{2i+1} := q_i$; $d_{2i} := s_i$; $d_{2i+1} := 0$ fi ;

if $i < 2^{s-1}$ then $t_i := u_{2i} + u_{2i+1}$ fi ;

if $i < 2^{s-1}$ then $u_i := t_i$ else $u_i := s_i$ fi

rof

It's clear that in this program for *Cell*(i) we can take the local variables s_i and t_i equal.

4 The All-Points Shortest Path Problem

We have given some weighted directed graph of n nodes. The weights of the edge from node i to node j is given as w_{ij}, where $w_{ij} \geq 0$. If there is no edge from i to j in the graph, then we add one with weight $w_{ij} = \infty$.

The problem is to find the shortest distance d_{ij} from node i to j for all i and j. A *path* from node i to node j is a sequence of nodes $(i_0 i_1 \ldots i_k)$ where $i_j \in 1..n$ for $j \in 0..k$ and where $i_0 = i$ and $i_k = j$. The *length* of a path $(i_0 i_1 \ldots i_k)$ is the number of edges k, whereas the distance along it (its *weight*) is $w_{i_0 i_1} + w_{i_1 i_2} + \cdots + w_{i_{k-1} i_k}$. Note that since all weights are non-negative the shortest path from i to j is well defined, it will never cross itself, and so has length smaller than n. We denote the collection of paths from i to j with length smaller than m by $path(i, j, m)$. Then the distance d_{ij} along the *shortest path* from i to j is defined as:

$$i = j \rightarrow d_{ij} = 0, \text{ and } i \neq j \rightarrow d_{ij} = \min\{w_{i_0 i_1} + w_{i_1 i_2} + \cdots + w_{i_{k-1} i_k}\},$$

where the minimum is over all sequences in $path(i, j, n)$. A simple fact that follows directly from the definition: If node p is on the shortest path from i to j, then the subpaths from i to p and from p to j are the *shortest* paths from i to p and from p to j. Moreover, $d_{ij} = d_{ip} + d_{pj}$. (For instance, if the subpath from i to p would *not* be the shortest one, then we could improve the $i - j$ path, contrary to the assumption that it was the shortest $i - j$ path.)

A well-known sequential algorithm for computing the shortest path, based on the above observations, is the Floyd-Warshall, cf. [Akl89], Chapter 10.

Program *Floyd-Warshall*

for $i \leftarrow [1 \ldots n]$ **doseq**

for $j \leftarrow [1 \ldots n]$ **doseq** $m[0,i,j] := w[i,j]$ **rof**

rof ;

for $k \leftarrow [1 \ldots n]$ **doseq**

for $i \leftarrow [1 \ldots n]$ **doseq**

for $j \leftarrow [1 \ldots n]$ **doseq** $m[k,i,j] := min\{m[k-1,i,j], m[k-1,i,k] + m[k-1,k,j]\}$ **rof**

rof

rof

An invariant of the above program is; $m[k,i,j]$ is the shortest path from node i to node j with all the intermediate nodes taken from $\{1,\ldots,k\}$. From which the correctness easily follows. Put

$$e_{i,k,j} = m[k,i,j] := min\{m[k-1,i,j], m[k-1,i,k] + m[k-1,k,j]\}$$

The Floyd-Warshall is a sequential algorithm where all actions are ordered in time. Again from point of functional correctness this is unnecessary; we can replace each sequential composition by a layer composition to get a process which is IO-equivalent to the initial process:

for $i \leftarrow [1 \ldots n]$ **layer**

for $j \leftarrow [1 \ldots n]$ **layer** $m[0,i,j] := w[i,j]$ **rof**

rof •

for $k \leftarrow [1 \ldots n]$ **layer**

for $i \leftarrow [1 \ldots n]$ **layer**

for $j \leftarrow [1 \ldots n]$ **layer** $\langle e_{i,k,j} \rangle$ **rof**

rof

rof

This process without sequential composition allows for algebraic transformations towards a specific architecture. Take for instance a **CREW-PRAM**, then there is only a conflict between $e_{i,k,j}$ and $e_{i,k',j}$. This yields, invoking the Left-Right Movers law, in that case the following normal form:

Program *Floyd-Warshall Normal Form I*

for $i \leftarrow [1 \ldots n]$ **dopar**

for $j \leftarrow [1 \ldots n]$ **dopar** $m[0,i,j] := w[i,j]$ **rof**

rof •

for $k \leftarrow [1 \ldots n]$ **layer**

for $i \leftarrow [1 \ldots n]$ **dopar**

for $j \leftarrow [1 \ldots n]$ **dopar** $\langle e_{i,k,j} \rangle$ **rof**

rof

rof

This leads to an implementation on a **CREW-PRAM** by changing every layer composition in a sequential composition, with complexity $O(n)$ time, on $O(n^2)$ processors. This is an optimal parallelization of the sequential Floyd-Warshall algorithm.

If we consider a **EREW-PRAM** then for fixed k $e_{i,k,j}$ is in conflict with $e_{i',k,j'}$ if and only if $i = i'$ or $j = j'$, because in that case they want to read both the same shared variable. The conflict order imposed by the sequential Floyd-Warshall algorithm is, again for fixed k

$$e_{i,k,j} \longrightarrow e_{i',k,j} \text{ iff } i < i' \text{ and } e_{i,k,j} \longrightarrow e_{i,k,j'} \text{ iff } j < j'$$

cf. figure 1, where one has to take the transitive closure in each row and column.

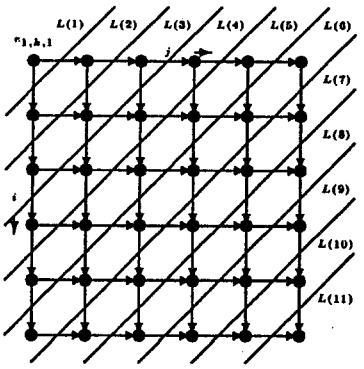

Figure 1: The causal ordering and layering for fixed k.

If we write this in pseudo normal form the computation becomes

Program *Floyd-Warshall Normal Form II*

for $i \leftarrow [1 \ldots n]$ **dopar**
  for $j \leftarrow [1 \ldots n]$ **dopar** $m[0,i,j] := w[i,j]$ **rof**
rof •
for $k \leftarrow [1 \ldots n]$ **layer**
  for $l \leftarrow [1 \ldots 2*n-1]$ **layer** $L(k,l)$ **rof**
rof

where for $l \leq n$

$$L(k,l) = \textbf{ for } j \leftarrow [1 \ldots l] \textbf{ dopar } \langle e_{j,k,l+1-j} \rangle \textbf{ rof}$$

and for $n < l \leq 2*n-1$

$$L(k,l) = \textbf{ for } j \leftarrow [1 \ldots 2*n-l] \textbf{ dopar } \langle e_{l-n+j,k,n+1-j} \rangle \textbf{ rof}$$

which leads to an algorithm with time complexity $O(n^2)$ and processor complexity $O(n)$ on an **EREW-PRAM**, which is again optimal.

Observe that the causal ordering on the $e_{i,k,j}$ for fixed k is only induced by Read/Read conflicts, which means that the choice of the ordering doesn't influence the total result of the computation. Hence we can take another minimal conflict closed ordering, for instance the one given in figure 2. Writing this computation in a pseudo normal form gives:

Program *Floyd-Warshall Normal Form III*

for $i \leftarrow [1 \ldots n]$ **dopar**
  for $j \leftarrow [1 \ldots n]$ **dopar** $m[0,i,j] := w[i,j]$ **rof**
rof •
for $k \leftarrow [1 \ldots n]$ **layer**
  for $l \leftarrow [1 \ldots n]$ **layer** $L'(k,l)$ **rof**
rof

where

$$L'(k,l) = \textbf{ for } i \leftarrow [1 \ldots n] \textbf{ dopar } \langle e_{i,k,i \oplus l} \rangle \textbf{ rof}$$

with

$$i \oplus l = (n + l - i) \bmod n + 1$$

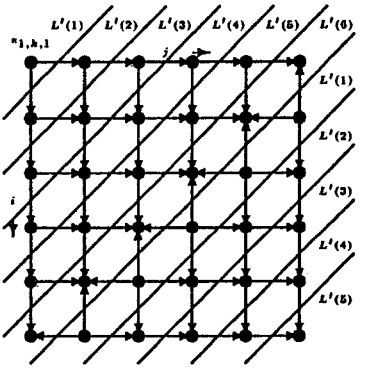

Figure 2: Another causal ordering for fixed k and the corresponding layering.

5 Conclusion

A language is presented which makes clear distinction between temporal order and causal order. This allows for a transformational approach, in an algebraic way, for the specification and verification of concurrent systems. Each process term without sequential composition can be transformed, in an algorithmic way, using the algebraic laws, into a normal form. This normal form is the maximal parallelization of the process under consideration. The techniques involved are exemplified by some examples from pipelining on a tree network, and the Floyd-Warshall algorithm for the all-points shortest path.

References

[Akl89] S.G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, 1989.

[AO91] K.R. Apt and E.-R. Olderog. *Verification of sequential and concurrent programs.* Springer-Verlag, 1991.

[EF82] Elrad and N. Francez. Decomposition of distributed programs into communication closed layers. *Science of Computer Programming*, 2, 1982.

[JPSZ91] W. Janssen, M. Poel, K. Sikkel, and J. Zwiers. The primordial soup algorithm: A systematic approach to the specification and design of parallel parsers. In *Proc. of CSN'91*, 1991.

[JPZ91] W. Janssen, M. Poel, and J. Zwiers. Action systems and action refinement in the development of parallel systems. In *Proc. of CONCUR '91*, pages 298–316. Springer-Verlag, LNCS 527, 1991.

[JZ92a] W. Janssen and J. Zwiers. From sequential layers to distributed processes, deriving a distributed minimum weight spanning tree algorithm. In *Proc. 11th PODC Symposium*, pages 215–227. ACM, 1992.

[JZ92b] W. Janssen and J. Zwiers. Protocol design by layered decomposition, a compositional approach. In *Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems.* Springer-Verlag, LNCS 571, 1992.

[PZ92] M. Poel and J. Zwiers. Layering techniques for development of parallel systems. Technical report, University of Twente, 1992. To appear.