THE PROJECTION OF SYSTOLIC PROGRAMS

C. LENGAUER[†,‡] AND J. W. SANDERS
PROGRAMMING RESEARCH GROUP
OXFORD UNIVERSITY COMPUTING LABORATORY
8–11 KEBLE ROAD, OXFORD, ENGLAND OX1 3QD

Abstract

A scheme is presented which transforms systolic programs with a two-dimensional structure to one dimension. The elementary steps of the transformation are justified by theorems in the theory of communicating sequential processes and the scheme is demonstrated with an example in occam: matrix composition/decomposition.

1 Introduction

We combine two types of formal refinement to transform a two-dimensional systolic program to one dimension. *Systolic arrays* are particularly regular distributed processor networks capable of processing large amounts of data quickly by accepting streams of input and producing streams of output [6]. Typical applications are to image or signal processing; ours is an algorithm which subsumes matrix composition and decomposition.

Systolic arrays are usually realized in hardware. We are interested in realizing them in *software*, because then they can run on one of the families of distributed computers (now plentiful) capable of emulating systolic arrays. We are led to express such software in a distributed programming language that provides constructs for process definition and communication. The production of that software is relatively straight-forward if the program's process and channel structure, which matches the processor and communication structure of the systolic array, also matches the distributed computer. That is not always the case. If the distributed computer does not offer the processor layout and interconnections that the systolic program prescribes, one has two options:

1. one can derive a systolic array that matches the limitations of the computer and derive a program from it, or
2. one can adjust the program derived from the ideal systolic array.

We pursue the second route, following the principle that real-world limitations should be imposed as late as possible in the design.

We consider one specific case: the processor layout of the machine has fewer dimensions than the process layout of the systolic program.[1] In this case, a *projection*, i.e., a transformation of

[†]On leave from the Department of Computer Sciences, The University of Texas at Austin, Taylor Hall 2.124, Austin, Texas 78712-1188.

[‡]Supported in part by the following funding agencies: through Oxford University by the Science and Engineering Research Council under Contract GR/E 63902; through the University of Texas at Austin by the Office of Naval Research under Contract N00014-86-K-0763, and by the National Science Foundation under Contract DCR-8610427.

[1]For work that explores the first option, see [7].

the process layout of the systolic program is required. We consider the transformation of two-dimensional systolic programs into one-dimensional systolic programs. There are programming environments that permit the specification of a mapping from software processes to hardware processors (e.g., for a Transputer network [4]), which makes explicit program projections unnecessary. We require this mapping to be the identity in order to avoid inefficiencies caused by the software simulation of channel communication.

The method we use to justify the projection from two to one dimension appears to be novel. It can be thought of as a variant of a hybrid of refinement techniques used in "formal methods". There, criteria for the refinement of *sequential* systems involve a relation between the states of the two systems [2,10]; criteria for the refinement of *concurrent* systems enable one system to be replaced by another in any environment [1,5]. We employ a technique of state relabelling which enables one system to replace another in any of a restricted class of environments. We hope this feature will be useful in other contexts. The refinement, as usual, makes a program more specific for the machine at hand: by postulating a one-dimensional systolic architecture, it leads from the ideal two-dimensional design to a one-dimensional implementation.

2 The Problem

We are given three matrices: A, B and C. Our goal is to establish that C is the matrix product of A and B:

$$(\forall\, i,j : 0 \le i,j < n : \quad c_{i,j} = (\sum k : 0 \le k < n : a_{i,k} \cdot b_{k,j}))$$

That goal may be achieved in different ways, depending on which of the matrices are to be determined. We consider two possibilities. Because we wish to derive a systolic solution we shall assume that the matrices are distinct program objects, i.e., they do not share elements.

2.1 Matrix Composition

A and B are input and C is output. A and B uniquely determine C.

2.2 Matrix Decomposition

C is input and A and B are output. For A and B to be determined uniquely, we require them to be triangular matrices: A is one on the diagonal and zero above it; B is zero below the diagonal.

3 The Two-Dimensional occam Program

A two-dimensional systolic occam program that establishes the required relation between A, B and C is listed in Appendix A.1. The program has been obtained by formal methods that are documented elsewhere [3,8]; we shall not justify its correctness here. We note some limitations of the original version of occam [4]:

Full Parenthesization. Arithmetic expressions must be fully parenthesized.

One-Dimensional Arrays Only. We must represent an $n \times n$ matrix by an $n * n$ vector. Read index `[(n*col)+row]` as index pair `[col,row]`.

No Floating-Point Arithmetic. We use a floating-point package. Read `RealOp(z,x,Op,y)` as `z := x Op y`.

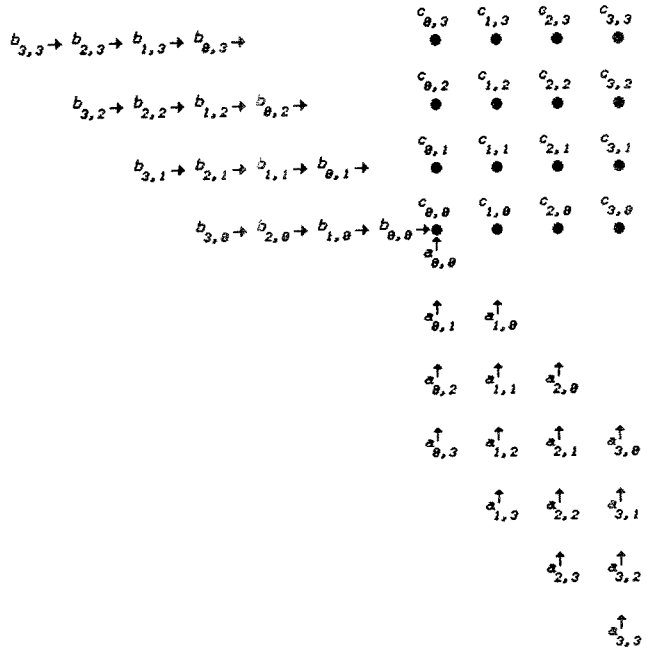

Figure 1: 4×4 Matrix Composition/Decomposition – The Two-Dimensional Systolic Array

A picture is helpful in understanding the structure of the program (see Fig. 1). Stream A moves through the processor array from bottom to top, stream B from left to right, and stream C is stationary during the computation (it must be loaded into the array before and recovered from it after the computation). The systolic program consists of three sets of processors (or cells):

Computation Cells. They first accept the stationary stream C from the left, then they execute the basic operations assigned to them, propagating streams A and B, and finally they eject stream C to the right.

Input Cells. Input cells on the left of the array inject first stream C and then stream B. Input cells on the bottom inject stream A.

Output Cells. Output cells on the right of the array extract first stream B and then stream C. Output cells on the top extract stream A.

Only the computation cells appear in Fig. 1, where they are represented by dots. The required channel connections can be inferred from the data flow. Fig. 1 indicates horizontal channels pointing right and vertical channels pointing up.

The program refers to a basic operation `BasicOp`. Its body differs for matrix composition and decomposition. We have not filled in the preprocessing and postprocessing phases. They also differ for matrix composition and decomposition. The following two subsections specify these individual refinements – for the sake of completeness. We shall not refer to them again.

3.1 Matrix Composition

For matrix composition, the basic operation is defined as follows:

```
PROC BasicOp(VALUE i, j, k, AElement, BElement, VAR CElement) =
  VAR tmp:
  SEQ
    RealOp(tmp, AElement, Mul, BElement)
    RealOp(CElement, CElement, Add, tmp) :
```

In the preprocessing phase, output matrix C is initialized to zero.

3.2 Matrix Decomposition

For matrix decomposition, the basic operation is defined as follows:

```
PROC BasicOp(VALUE i, j, k, VAR AElement, BElement, CElement) =
  VAR tmp:
  SEQ
    IF
      (i<=j) AND (i=k)
        BElement := CElement
      (i>j) AND (j=k)
        SEQ
          RealOp(tmp, One, Div, BElement)
          RealOp(AElement, CElement, Mul, tmp)
      (i>k) AND (j>k)
        SEQ
          RealOp(tmp, AElement, Mul, BElement)
          RealOp(CElement, CElement, Sub, tmp)
      TRUE
        SKIP :
```

In the preprocessing phase, the output matrices A and B are initialized to the identity and zero, respectively.

4 The Projection

We eliminate the vertical dimension by projecting horizontally. In accordance with Fig. 1, we shall refer to the three data streams as follows:

A is the *projected stream.* Its direction of flow is in the dimension that disappears in the projection. A is turned from a moving into a stationary stream.

B is the *moving stream.* It remains moving to the right.

C is the *stationary stream.* It remains stationary.

We perform the projection in two steps: we combine first the cells and then the channels of each column into one. Both steps result in startlingly simple program transformations as far as the moving and stationary streams are concerned. The cell projection of the handling of the projected stream is more complicated: it involves a conversion from moving to stationary

and – more seriously – a redirection of the stream i/o. We provide first an informal account of the projection, then several transformation theorems and, finally, their application in the transformation. The reader may find it helpful to consult the appendix throughout the following subsections.

4.1 Informal Description

4.1.1 Cell Projection

The Moving and Stationary Streams

We replace the `PAR` loop over the dimension that is projected away by a `SEQ` loop. In our case, it is the dimension indexed by `row` (App. A.1, line 20 and App. A.2, line 30). This combines the computation processes for each column in increasing sequence rather than in parallel. We account similarly for the projection by replacing `SEQ` for `PAR` in the input and output loops on `row` (App. A.1, lines 13 and 44 and App. A.2, lines 13 and 59). Also, since variables `BElement` in each column of computation cells are now being accessed in sequence without overlap, we represent them by a single variable: we move the declaration of `BElement` from the loop on `row` out to the loop on `col` (App. A.1, line 21 and App. A.2, line 20).

The Projected Stream

For loading and recovery, we convert the flow direction of the stream from vertical to horizontal by commuting `col` and `row` in the input and output loops for the projected stream and replacing channels `Up` by channels `Right` (App. A.1, lines 10–12 and 50–52 and App. A.2, lines 10–12 and 65–67). We must also eliminate the communications on `Up` in the computation processes (App. A.1, lines 32 and 36). Then we account for the projection by replacing `SEQ` for `PAR`, now for the loop on `row` (App. A.2., lines 10 and 65). We also add a loading and recovery phase to the computation processes (App. A.2, lines 22–29 and 50–57). Each process must hold the stationary elements of one column of the array. We convert variable `AElement` into a vector and declare it per column of the array instead of per column and row (App. A.1, line 21 and App. A.2, line 20).

4.1.2 Channel Projection

We simply discard the dimension that is projected away – here it is `row` – from the channel array.

4.2 Theorems

We reason in a language $\mathcal{P}$, which lies midway between CSP [1] and the restricted subset of occam [11] used to express our programs. It includes those processes which engage in a finite number of inputs, outputs and assignments before terminating. From CSP, it inherits a calculus of communication traces and refusals; from occam, it inherits local variables. Since we do not consider infinite or divergent programs, we are able to reason using a drastically simplified semantics.

Each process P is described by

- a channel alphabet γP (those channels on which P may communicate),
- a variable alphabet νP (P's local program variables),

- communications (via P's failures; see [1, Chap. 3]), and
- the change in program variables (which we describe by using a predicate whose free variables consist of the vector x of P's variable values before execution and the vector x' of P's variable values after execution).

There are three types of basic process in $\mathcal{P}$. We now describe each informally and say how their variables change; it is implicit that a process cannot change variables outside its alphabet. For a description of their refusals, the reader is referred to [1, Chap. 3]. There, slightly different syntax is used: each basic process is regarded as an event and is converted to a process by postfixing it with *SKIP*. From [1, Chap. 1], we also adopt the notation P **sat** S, which means that process P satisfies condition S.

An input process $P = c?x$ inputs the value e communicated on channel c and assigns it to variable x. Its alphabet has $\gamma P = \{c\}$ and $\nu P = \{x\}$ and, regardless of its previous value, the final value x' of x equals e

$$P \textbf{ sat } x' = e.$$

An output process $P = c!e$ outputs the value of expression e on channel c. Its alphabet has $\gamma P = \{c\}$ and $\nu P = \{\}$, so it cannot alter any variables.

An assignment process $P = x{:=}e$ assigns the value of expression e to variable x. Its alphabet has $\gamma P = \{\}$ and $\nu P = \{x\}$; it achieves the same program state as the previous input process, but without any communications:

$$P \textbf{ sat } x' = e.$$

Processes are combined using sequential composition, denoted $\rightarrow$, and parallel composition, denoted $\|$. When a pair of processes is being composed, we use these symbols in infix; for the composition of a sequence $(i : 0 \leq i < n : P_i)$ of processes, we use the prefix notations $(\rightarrow i : 0 \leq i < n : P_i)$ and $(\| i : 0 \leq i < n : P_i)$. Again, we refer to [1] for laws satisfied by sequential and parallel composition (there, the semicolon is used for sequential composition; we reserve that symbol for forward composition of binary relations and predicates). As usual, we suppose that processes are only placed in parallel if none accesses a variable that another modifies, thus

$$\gamma(\| i : 0 \leq i < n : P_i) = (\cup\, i : 0 \leq i < n : \gamma P_i)$$

$$\nu(\| i : 0 \leq i < n : P_i) = (\cup\, i : 0 \leq i < n : \nu P_i)$$

and, if S_i is a predicate in the variables x and x' of P_i with

$$(\forall\, i : 0 \leq i < n : P_i \textbf{ sat } S_i),$$

then

$$(\| i : 0 \leq i < n : P_i) \textbf{ sat } (\forall\, i : 0 \leq i < n : S_i).$$

For sequential composition, no constraint on variable accesses applies, of course, thus

$$\gamma(\rightarrow i : 0 \leq i < n : P_i) = (\cup\, i : 0 \leq i < n : \gamma P_i)$$

$$\nu(\rightarrow i : 0 \leq i < n : P_i) = (\cup\, i : 0 \leq i < n : \nu P_i)$$

and, if S_i is a predicate in the variables of P_i with

$$(\forall\, i : 0 \leq i < n : P_i \textbf{ sat } S_i),$$

then

$$(\rightarrow i : 0 \leq i < n : P_i) \textbf{ sat } (;i : 0 \leq i < n : S_i).$$

For example, if P and Q are processes with

$$\nu P = \{x, y\} \text{ and } \nu Q = \{y, z\}$$

such that

$$P \textbf{ sat } x' = f(x) \wedge y' = g(y)$$
$$Q \textbf{ sat } y' = h(y) \wedge z' = k(z)$$

then

$$\nu(P \rightarrow Q) = \{x, y, z\}$$
$$(P \rightarrow Q) \textbf{ sat } x' = f(x) \wedge y' = h(g(y)) \wedge z' = k(z).$$

When considering a parallel composition, we shall often stress one process by referring to the other(s) as its environment. When we say that P satisfies property S in environment Q, we mean

$$(P \parallel Q) \textbf{ sat } S.$$

Many of our transformations replace one process with another, in a given environment.

We shall use the law

$$((c!e \rightarrow P) \parallel (c?x \rightarrow Q)) \;=\; (c!e \rightarrow x{:=}e \rightarrow (P \parallel Q)). \tag{1}$$

Using this law, and those from [1,11], we reason in $\mathcal{P}$ about **occam** programs, just as one reasons in the language of guarded commands about Modula-2 programs. Down-coding is done by identifying the basic processes with **occam** programs (from [11, Sect. 2]):

$c?x$	with	`VAR` y : `ALT` $(c?y \;\; x{:=}x[y/x])$
$c!e$	with	`ALT` $(c!e \;\; x{:=}x)$
$x{:=}e$	with	$x{:=}x[e/x]$

and by identifying $\parallel$ with `PAR` and $\rightarrow$ with `SEQ`.

The final operation we require is that of concealment. If $E \subseteq \gamma P$, then $P \setminus E$ is a process which behaves like P but with all communications on channels in E concealed. Thus $\gamma(P \setminus E) = (\gamma P) \setminus E$ and $\nu(P \setminus E) = \nu P$, and no variables are altered by $P \setminus E$. For the failures of $P \setminus E$ and for the laws satisfied by concealment, see [1, Chap. 3].

In the following subsections, we justify all transformations except the movement of the declaration of variable `BElement`.

4.2.1 Cell Projection

We use three theorems. The first, the cell projection theorem, takes care of moving and stationary streams. Two more theorems, the stream projection and stream reflection theorem, address the treatment of projected streams.

The Moving and Stationary Streams

The cell projection theorem addresses two properties of a finite set of messages, which are communicated over separate channels:

1. Messages that are consumed in a total order may be produced in the same order or in any approximating (i.e., less defined) order.
2. Messages that are produced in some partial order may be consumed in that partial order or in any more defined order, provided the target variables are distinct.

For our purposes, a more restricted version of the cell projection theorem suffices: it takes for the approximating order the undefined order (which relates no elements at all) and for the approximated order a total order (which relates all elements).

Cell Projection Theorem:

Let c_i be distinct channels, e_i expressions and x_i distinct variables. The processes

$$(\|\ i : 0 \le i < n : c_i?x_i) \quad \text{and} \quad (\rightarrow i : 0 \le i < n : c_i?x_i)$$

both satisfy condition $S = (\forall\, i : 0 \le i < n : x_i' = e_i)$ in either of the environments

$$(\|\ i : 0 \le i < n : c_i!e_i) \quad \text{or} \quad (\rightarrow i : 0 \le i < n : c_i!e_i).$$

Proof:

There are four processes to consider; we enumerate them for ease of reference (in fact, only the first three appear in our application). Even though their communication traces are not the same, the final state of all four processes are.

(a)
$$\begin{array}{l} \quad (\|\ i : 0 \le i < n : c_i!e_i) \parallel (\|\ i : 0 \le i < n : c_i?x_i) \\ = \{\text{by [1, Sect. 4.3: L1] and (1)}\} \\ \quad (\|\ i : 0 \le i < n : c_i!e_i \rightarrow x_i{:=}e_i) \\ \textbf{sat} \\ \quad S. \end{array}$$

(b)
$$\begin{array}{l} \quad (\|\ i : 0 \le i < n : c_i!e_i) \parallel (\rightarrow i : 0 \le i < n : c_i?x_i) \\ = \{\text{by [1, Sect. 2.3.1: L7, Sect. 4.3: L1] and (1)}\} \\ \quad (\rightarrow i : 0 \le i < n : c_i!e_i \rightarrow x_i{:=}e_i) \\ \textbf{sat} \\ \quad S. \end{array}$$

The proofs that

(c) $(\rightarrow i : 0 \le i < n : c_i!e_i) \parallel (\rightarrow i : 0 \le i < n : c_i?x_i)$ **sat** S

and

(d) $(\rightarrow i : 0 \le i < n : c_i!e_i) \parallel (\|\ i : 0 \le i < n : c_i?x_i)$ **sat** S

are identical to (b).

End of Proof.

The Projected Stream

Two things happen in the transformation of a projected stream:

1. "moving" is projected to "stationary"; this affects elements that are processed in sequence by one column,
2. "up" is reflected to "right"; this affects elements that are processed in parallel by different columns.

Stream Projection Theorem:

Let c_i be distinct channels, f_i expressions (i.e., functions) in one parameter, x_i distinct variables, x a variable and let $f = (;\, i : 0 \leq i < n : f_i)$ denote the forward relational composition of the f_i. Then, condition $z' = f(e)$ is satisfied by both the processes

$$P \;=\; (\|\, i : 0 \leq i < n : c_i?x_i \rightarrow x_i{:=}f_i(x_i) \rightarrow c_{i+1}!x_i)\backslash\{i : 0 < i < n : c_i\}$$

and

$$Q \;=\; c_0?x \rightarrow (\rightarrow i : 0 \leq i < n : x{:=}f_i(x)) \rightarrow c_n!f(x)$$

in environment $c_0!e \rightarrow c_n?z$.

Proof:

In spite of the fact that P and Q have the same communication traces, they are not equal since they have different alphabets. However, by a repeated application of [1, Sect. 3.5.1: L5],

$$P \;=\; c_0?x_0 \rightarrow (\rightarrow i : 0 < i < n : x_{i+1}{:=}f_i(x_i)) \rightarrow c_n!x_n$$

hence

$$P \parallel (c_0!e \rightarrow c_n?z) \;\; \textbf{sat} \;\; z' = f(e).$$

The result for Q is immediate.

End of Proof.

The reflection is more intricate to express, because it is a conversion of the vertical-parallel i/o of stream values into a horizontal-sequenced i/o with loading and recovery.

Stream Reflection Theorem:

Let c_i and d_i be distinct channels, f_i expressions in one parameter and x_i, xin_i, $xout_i$ and $xtmp_i$ distinct variables. Condition $S = (\forall\, i : 0 \leq i < n : xout_i' = f_i(xin_i))$ is satisfied by the following processes in their respective environments:

(a) by process

$$(\|\, i : 0 \leq i < n : c_i?x_i \rightarrow x_i{:=}f_i(x_i) \rightarrow d_i!x_i)$$

in environment

$$(\|\, i : 0 \leq i < n : c_i!xin_i) \parallel (\|\, i : 0 \leq i < n : d_i?xout_i),$$

(b) by process

$$\begin{array}{lll}(\|\ i : 0 \le i < n : & c_i?x_i & \\ & \rightarrow\ (\rightarrow j : i < j < n : c_i?xtmp_i \rightarrow c_{i+1}!xtmp_i) & \\ & \rightarrow\ x_i{:=}f_i(x_i) & \\ & \rightarrow\ (\rightarrow j : 0 \le j < i : c_i?xtmp_i \rightarrow c_{i+1}!xtmp_i) & \\ & \rightarrow\ c_{i+1}!x_i) & \end{array}$$

in environment

$$(\rightarrow i : 0 \le i < n : c_0!xin_i) \parallel (\rightarrow i : 0 \le i < n : c_n?xout_i).$$

Proof.

(a) By [1, Sect. 2.3.1: L7, Sect. 4.3: L1] and (1), this process simplifies to

$$(\|\ i : 0 \le i < n : c_i!xin_i \rightarrow x_i{:=}f_i(xin_i) \rightarrow d_i!f_i(xin_i) \rightarrow xout_i{:=}f_i(xin_i))\ \ \textbf{sat}\ \ S$$

(b) The resulting process cannot be simplified like the previous one (because of its communications) but nevertheless satisfies the condition (which involves only variables) by induction. In the case $n = 1$,

$$\begin{array}{l} \quad c_0!xin_0 \parallel (c_0?x_0 \rightarrow x_0{:=}f_0(x_0) \rightarrow c_1!x_0) \parallel c_1?xout_0 \\ = \{\text{by [1, Sect. 2.3.1: L7, Sect. 4.3: L1] and (1)}\} \\ \quad c_0!xin_0 \rightarrow x_0{:=}f_0(xin_0) \rightarrow c_1!f_0(xin_0) \rightarrow xout_0{:=}f_0(xin_0) \\ \textbf{sat} \\ \quad xout'_0 = f_0(xin_0). \end{array}$$

For the induction step, assume the result for n. Then, for $n+1$, the i^{th} process satisfies, after $n+1-i$ inputs on channel c_i, $xtmp'_i = e$, where e is the $n+1-i^{th}$ value input on that channel. Thus, the process in its environment satisfies

$$(\forall\, i : 0 \le i < n : xout'_i = f_i(xin_i)) \wedge xout'_n = f_n(xin_n)$$

as required.

End of Proof.

4.2.2 Channel Projection

The channel projection theorem states that successive communications on separate channels can be transmitted over a common channel.

Channel Projection Theorem:

Let c_i be distinct channels and c a channel, e_i expressions and x_i distinct variables. Condition $S = (\forall\ i : 0 \le i < n : x'_i = e_i)$ is satisfied by the following processes in their respective environments:

(a) process $(\rightarrow i : 0 \le i < n : c_i?x_i)$ in environment $(\rightarrow i : 0 \le i < n : c_i!e_i)$,

(b) process $(\rightarrow i : 0 \le i < n : c?x_i)$ in environment $(\rightarrow i : 0 \le i < n : c!e_i)$.

Proof.

(a) Case (c) of cell projection.

(b) Simply replace c_i by c in the proof above since the condition is independent of channel names.

End of Proof.

4.3 Application

4.3.1 Cell Projection

The Moving and Stationary Streams

The cell projection theorem is applied recursively to each column of cells, right to left. As the base case, consider the output cells. A substitution of the `PAR` on `row` by `SEQ` (App. A.1, line 44 and App. A.2, line 59) is a conversion from (a) to (b) in the theorem. The induction hypothesis is that the input of stationary and moving streams in some column has been projected. Consider the column to its left. In the induction step, we project the output of the stationary and moving streams in that column by converting from (b) to (c), and the input of stationary and moving streams in that column by converting from (a) to (b) again. This means replacing the `PAR` on `row` that encases both the input and the output of stationary and moving streams by a `SEQ` (App. A.1, line 20 and App. A.2, line 30). This is done for a fixed iteration of the encasing `PAR` loop on `col` (App. A.1 and A.2, line 19). The induction is on `col`. Finally, we replace the `PAR` on `row` in the column of input cells by a `SEQ` (App. A.1 and A.2, line 13), converting once more from (b) to (c).

It is important to note that we apply the cell projection theorem in a benevolent environment: no communications other than the ones stated in the theorem are transmitted over the stated channels; in particular, there are no messages in the reverse direction, i.e., there is no feedback that could create circular dependencies.

We do not justify the projection of variable `BElement` formally. The theory of CSP, as presented in [1], does not provide for this kind of program transformation.

The Projected Stream

Initially, we had entertained the hope that we could split the cell projection of the treatment of stream A into two steps:

1. the reflection of the stream from horizontal to vertical and its treatment as stationary, and

2. the cell projection, as previously for stationary streams.

Unfortunately, the intermediate (still two-dimensional) program – the output of step 1 and input of step 2 – is not occam. It is dangerous to pretend that the projected stream is stationary in the two-dimensional array when it really is not. Since cells share the stream's elements, they update shared variables, which is illegal in occam.

The stream projection and stream reflection theorem split the projection of stream A up differently. We have to perform the cell projection first in order to avoid the ill-defined intermediate two-dimensional program. Our transformation proceeds as follows.

By appealing to the stream projection theorem, we eliminate lines 32 and 36 of the two-dimensional program. These communications on `Up` correspond to the communications on the middle channels $c_1, ..., c_{n-1}$ in the theorem. We apply the theorem by substituting Q for P. The transformation of the variables x_i in P to the single variable x in Q corresponds to the movement of the declaration of variable `AElement` from the `PAR` on `row` in the two-dimensional program (line 21) to the `PAR` loop on `col` in the one-dimensional program (line 20). `AElement` becomes a vector of variables as a result of an inductive application of the stream projection theorem.

By appealing to the stream reflection theorem, we frame the computation cell code with code portions for the loading and recovery of stationary stream A (App. A.2, lines 22–29 and 50–57). We convert from (a) to (b). The stream reflection theorem also covers the reflection of the input and output from channels `Up` in the two-dimensional program (App. A.1, lines 10–12 and 50–52) to channels `Right` in the one-dimensional program (App. A.2, lines 10–12 and 65–67).

4.3.2 Channel Projection

The channel projection theorem is also applied recursively, converting from (a) to (b). Each application collapses the set of channels between two columns (which have already been collapsed to single cells) to a single channel. At this point, there are only stationary and moving streams left; no distinction needs to be made between them.

5 Conclusions

We have formulated a general projection scheme of cells and channels in systolic programs and have applied it to a specific example. We have stated and proved the transformation theorems in a language $\mathcal{P}$, and our scheme applies to any distributed programming language that obeys $\mathcal{P}$'s laws. The program example is in one implemented programming language that obeys those laws: occam. At present, there is a gap between the theorems and their application; we have bridged it with English explanations. Making our transformations completely precise is possible but elaborate. It would be easier were we to replace the mechanical systolic design that produces our two-dimensional occam program by a derivation that relies more on the semantics of processes in $\mathcal{P}$ – future work. Such a derivation would also simplify adjustments like the elimination of the `PAR` on lines 31 and 35 in the two-dimensional program together with the `Up` communications on lines 33 and 36 (`PAR`s with just one statement can be dropped).

Similar projection schemes can be applied to systolic programs with different stream movements. We have covered the three elementary cases: projected, moving and stationary streams. All of the required substitutions are static and can be easily incorporated into a compiler. We have tried to make the substitutions as simple as possible. The simplicity of the substitutions depends on the form of the program to be projected.[2] Luckily, our two-dimensional program is itself the product of a mechanical derivation [8]. Thus, we have the choice of imposing a derivation scheme that simplifies projections.

Our theorems are simplified by the fact that we distinguish the computation cells from the i/o cells, and that we reason about states rather than traces wherever possible. In the application of the theorems, we presume the absence of feedback between the computation and the i/o cells. Programs that are compiled from linear systolic designs [8] do not have feedback, but

[2]For example, letting computation cells during the loading of stationary streams propagate elements first and then keep their own element leads to an inversion of the column's index in the input loop, which complicates the projection of stream A.

feedback may arise when a linear systolic design is partitioned, i.e., transformed into a ring or toroid (e.g., [9]). Therefore, to keep the transformations simple, we suggest to project first and partition later.

6 Acknowledgements

The first author is indebted to Tony Hoare for the invitation to join PRG for a term, and to all members of PRG for making his stay a fruitful, enjoyable, and unforgettable one. Thanks go also to Richard Miller, Andrew Kay, and Geraint Jones, who assisted in the use of the occam language and compiler.

7 References

[1] C. A. R. Hoare, *Communicating Sequential Processes*, Series in Computer Science, Prentice-Hall Int., 1985.

[2] C. A. R. Hoare, He Jifeng and J. W. Sanders, "Prespecification in Data Refinement", *Information Processing Letters 25*, 2 (May 1987), 71–76.

[3] C.-H. Huang and C. Lengauer, "The Derivation of Systolic Implementations of Programs", *Acta Informatica 24*, 6 (Nov. 1987), 595–632.

[4] INMOS Ltd., occam *Programming Manual*, Series in Computer Science, Prentice-Hall Int., 1984.

[5] J. L. Jacob, "On Shared Systems", D. Phil. Thesis, Programming Research Group, Oxford University Computing Laboratory, 1987.

[6] H. T. Kung and C. E. Leiserson, "Algorithms for VLSI Processor Arrays", in *Introduction to VLSI Systems*, C. Mead and L. Conway (eds.), Addison-Wesley, 1980, Sect. 8.3.

[7] P. Lee, Z. Kedem, "Synthesizing Linear Array Algorithms from Nested for Loop Algorithms", *IEEE Trans. on Computers TC-37*, 12 (Dec. 1988), 1578-1598.

[8] C. Lengauer, "Towards Systolizing Compilation: An Overview", Proc. *Conf. on Parallel Architectures and Languages Europe (PARLE 89)*, June 1989, to appear as Springer-Verlag Lecture Notes in Computer Science.

[9] D. I. Moldovan and J. A. B. Fortes, "Partitioning and Mapping Algorithms into Fixed-Size Systolic Arrays", *IEEE Trans. on Computers C-35*, 1 (Jan. 1986), 1–12.

[10] T. Nipkow, "Non-Determinstic Data Types", *Acta Informatica 22*, 6 (Mar. 1986), 629–661.

[11] A. W. Roscoe and C. A. R. Hoare, "The Laws of occam Programming", *Theoretical Computer Science 60*, 2 (1988), 177ff.

A The occam Programs

A.1 The Two-Dimensional Program

```
1   VAR AIn[n*n], AOut[n*n],
2       BIn[n*n], BOut[n*n],
3       CIn[n*n], COut[n*n]:

4   CHAN Right[n*(n+1)]:          -- horizontal channels
5   CHAN Up[(n+1)*n]:             -- vertical channels

6   SEQ

      -- PREPROCESSING

7     "read in input matrices"
8     "initialize output matrices"

9     PAR

        -- INPUT CELLS

        -- vertical input: inject stream A

10      PAR col = [0 FOR n]
11        SEQ row = [0 FOR n]
12          Up[((n+1)*col)+0] ! AIn[(n*col)+row]

        -- horizontal input: load stream C and inject stream B

13      PAR row = [0 FOR n]
14        SEQ

15          SEQ col = [0 FOR n]
16            Right[(n*0)+row] ! CIn[(n*col)+row]

17          SEQ col = [0 FOR n]
18            Right[(n*0)+row] ! BIn[(n*col)+row]

        -- COMPUTATION CELLS

19      PAR col = [0 FOR n]
20        PAR row = [0 FOR n]

21          VAR AElement, BElement, CElement:
```

```
22          SEQ

              -- load stream C

23            Right[(n*col)+row] ? CElement

24            SEQ unused = [0 FOR (n-1)-col]
25              VAR tmp:
26              SEQ
27                Right[(n*col)+row] ? tmp
28                Right[(n*(col+1))+row] ! tmp

              -- do the computation

29            SEQ k = [0 FOR n]
30              SEQ

31                PAR
32                  Up[((n+1)*col)+row] ? AElement
33                  Right[(n*col)+row] ? BElement

34                BasicOp(col, row, k, AElement, BElement, CElement)

35                PAR
36                  Up[((n+1)*col)+row+1] ! AElement
37                  Right[(n*(col+1))+row] ! BElement

              -- recover stream C

38            SEQ k = [0 FOR col]
39              VAR tmp:
40              SEQ
41                Right[(n*col)+row] ? tmp
42                Right[(n*(col+1))+row] ! tmp

43            Right[(n*(col+1))+row] ! CElement

        -- OUTPUT CELLS

        -- horizontal output: extract stream B and recover stream C

44      PAR row = [0 FOR n]
45        SEQ

46          SEQ col = [0 FOR n]
47            Right[(n*n)+row] ? BOut[(n*col)+row]

48          SEQ col = [0 FOR n]
49            Right[(n*n)+row] ? COut[(n*col)+row]
```

```
        -- vertical output: extract stream A

50      PAR col = [0 FOR n]
51        SEQ row = [0 FOR n]
52          Up[((n+1)*col)+n] ? AOut[(n*col)+row]

      -- POSTPROCESSING

53    "read out output matrices"
```

A.2 The One-Dimensional Program

```
1   VAR AIn[n*n], AOut[n*n],
2       BIn[n*n], BOut[n*n],
3       CIn[n*n], COut[n*n]:

4   CHAN Right[n+1]:

5   SEQ

      -- PREPROCESSING

6     "read in input matrices"
7     "initialize output matrices"

8     PAR

        -- INPUT CELLS

9       SEQ

          -- load stream A

10        SEQ row = [0 FOR n]
11          SEQ col = [0 FOR n]
12            Right[0] ! AIn[(n*col)+row]

          -- load stream C and inject stream B

13        SEQ row = [0 FOR n]
14          SEQ

15            SEQ col = [0 FOR n]
16              Right[0] ! CIn[(n*col)+row]

17            SEQ col = [0 FOR n]
18              Right[0] ! BIn[(n*col)+row]
```

```
        -- COMPUTATION CELLS
19      PAR col = [0 FOR n]
20        VAR AElement[n], BElement:
21        SEQ
            -- load stream A
22          SEQ row = [0 FOR n]
23            SEQ
24              Right[col] ? AElement[row]
25              SEQ unused = [0 FOR (n-1)-col]
26                VAR tmp:
27                SEQ
28                  Right[col] ? tmp
29                  Right[col+1] ! tmp
30          SEQ row = [0 FOR n]
31            VAR CElement:
32            SEQ
                -- load stream C
33              Right[col] ? CElement
34              SEQ unused = [0 FOR (n-1)-col]
35                VAR tmp:
36                SEQ
37                  Right[col] ? tmp
38                  Right[col+1] ! tmp
                -- do the computation
39              SEQ k = [0 FOR n]
40                SEQ
41                  Right[col] ? BElement
42                  BasicOp(col, row, k, AElement[k], BElement, CElement)
43                  Right[col+1] ! BElement
```

```
                -- recover stream C

44              SEQ unused = [0 FOR col]
45                VAR tmp:
46                SEQ
47                  Right[col] ? tmp
48                  Right[col+1] ! tmp

49              Right[col+1] ! CElement

            -- recover stream A

50          SEQ row = [0 FOR n]
51            SEQ

52              SEQ unused = [0 FOR col]
53                VAR tmp:
54                SEQ
55                  Right[col] ? tmp
56                  Right[col+1] ! tmp

57              Right[col+1] ! AElement[row]

        -- OUTPUT CELLS

58      SEQ

          -- extract stream B and recover stream C

59        SEQ row = [0 FOR n]
60          SEQ

61            SEQ col = [0 FOR n]
62              Right[n] ? BOut[(n*col)+row]

63            SEQ col = [0 FOR n]
64              Right[n] ? COut[(n*col)+row]

          -- recover stream A

65        SEQ row = [0 FOR n]
66          SEQ col = [0 FOR n]
67            Right[n] ? AOut[(n*col)+row]

      -- POSTPROCESSING

68    "read out output matrices"
```