In S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, editors,
Handbook of Signal Processing Systems, pages 899-930. Springer, 2010.

Dynamic and Multidimensional Dataflow Graphs

Shuvra S. Bhattacharyya, Ed F. Deprettere, and Joachim Keinert

Abstract Much of the work to date on dataflow models for signal processing system design has focused decidable dataflow models that are best suited for one-dimensional signal processing. In this chapter, we review more general dataflow modeling techniques that are targeted to applications that include multidimensional signal processing and dynamic dataflow behavior. As dataflow techniques are applied to signal processing systems that are more complex, and demand increasing degrees of agility and flexibility, these classes of more general dataflow models are of correspondingly increasing interest. We begin with a discussion of two dataflow modeling techniques — multi-dimensional synchronous dataflow and windowed dataflow — that are targeted towards multidimensional signal processing applications. We then provide a motivation for dynamic dataflow models of computation, and review a number of specific methods that have emerged in this class of models. Our coverage of dynamic dataflow models in this chapter includes Boolean dataflow, the stream-based function model, CAL, parameterized dataflow, and enable-invoke dataflow.

1 Multidimensional synchronous data flowgraphs (MDSDF)

In many signal processing applications, the tokens in a stream of tokens have a dimension higher than one. For example, the tokens in a video stream represent images so that a video application is actually three-dimensional. Static multidimensional

Shuvra S. Bhattacharyya
University of Maryland, USA e-mail: ssb@umd.edu

Ed F. Deprettere
Leiden University, The Netherlands e-mail: edd@liacs.nl

Joachim Keinert
Fraunhofer Institute for Integrated Circuits, Germany e-mail: joachim.keinert@iis.fraunhofer.de

(MD) streaming applications can be modeled using unidimensional dataflow graphs (see Chapter 30), but these are at best cyclo-static dataflow graphs, often with many phases in the actor's vector valued token production and consumption patterns. These models incur a high control overhead that can be avoided when the application is modeled in terms of a multidimensional dataflow graph that may turn out to be just a multidimensional version of the unidimensional synchronous dataflow graph. Such graphs are called *multidimensional synchronous dataflow graphs* (MDSDF).

1.1 Basics

MDSDF is — in principle — a straightforward extension of SDF (see Chapter 30). Fig. 1 shows a simple example.

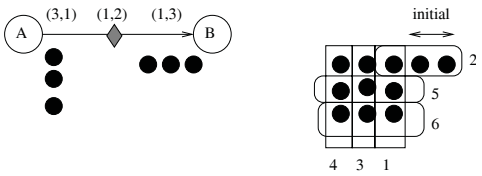


Fig. 1 A simple two-actor MDSDF graph.

Actor A produces a (3, 1) token which is a shorthand for *3 rows* and *1 column* out of an array of elements, and actor B consumes (1, 3) token (1 row and 3 columns). Note that only one dimension can be the streaming dimension. In the example of Fig. 1, this is the horizontal (column) dimension. The SDF 1D repetitions vector (see Chapter 30) becomes a *repetitions matrix* in the MDSDF case.

$$R = \begin{pmatrix} r_{A,1} & r_{B,1} \\ r_{A,2} & r_{B,2} \end{pmatrix}$$

where $r_{A,1} \times 3 = r_{B,1} \times 1$, and $r_{A,2} \times 1 = r_{B,2} \times 3$. Thus actor A has to produce a number of times (3,1) tokens, and actor B has to consume a number of times (1,3) tokens. The equations can be solved to yield $(r_{A,1}, r_{A,2}) = (1,3)$, and $(r_{B,1}, r_{B,2}) = (3,1)$. In other words, actor A has to produce once three rows, and three times one column, and similarly for actor B. These equations are independent of initial token tuples (x,y) in the arc's buffer. In Fig. 1, the buffer contains initial tokens in two columns of a single row, a *delay* of (1,2) that is. The production/consumption cycle goes in the order of the numbers given at the right in the figure. Note that the state returns to its initial value (1,2).

1.2 Arbitrary sampling

Two canonical actors that play a fundamental role in MD signal processing are the *compressor* (or downsampler) and the *expander* (or upsampler). An MD signal $s(n), n \in \mathbb{R}^m$ [1], is defined on a raster, or a *lattice*. A raster is generated by the columns of a (non-singular) *sampling matrix* V. For example, in the 2D case, the sampling matrix

$$V = \begin{pmatrix} v_{1,1} & 0 \\ 0 & v_{2,2} \end{pmatrix}$$

yields a *rectangular* sampling raster. Similarly, the sampling matrix

$$V = \begin{pmatrix} v_{1,1} = v_1 & v_{1,2} = v_2 \\ v_{2,1} = v_1 & v_{2,2} = -v_2 \end{pmatrix}$$

yields a *hexagonal* raster, as shown in the upper part of Fig. 2, where only a finite *k-support* defined by the columns of the matrix $P = diag(7,6)$ is displayed. Thus, $n = Vk,\ k \in \{\mathscr{P} \cap \mathit{F}^2\}$, where $\mathscr{P}$ is the parallelepiped defined by the columns of the matrix P. Notice that for an integral sampling matrix V, the number of integer points in $\mathscr{V}$ — denoted $N(V)$ — is equal to $\mid det(V) \mid$. $\mathscr{V}$ for the rectangular and the hexagonal sampling matrix examples in the upper part of Figure 2 are shown as shaded parallelepipeds.

Sampling lattices can be arbitrary which make MDSDF graphs more complicated than their unidimensional SDF counterparts.

The black dots in Figure 2 are sample points $n = Vk,\ k \in \{\mathscr{P} \cap \mathbb{Z}^2\}$.

The MD downsampler is characterized by an integral matrix M which takes sample points n to sample points $m = n$ for all $n = V_I Mk$, discarding all other points. Here, V_I is the sampling matrix at the input of the downsampler. The sampling matrix at the output is $V_O = V_I M$. The values of signal samples are carried over from input to output at the retained sample points. Similarly, the MD upsampler is characterized by an integral matrix L which takes sample points n to sample points $m = n$ for all $n = V_I k$, with equal sample values, and creates additional sample points $m = V_I L^{-1} k$ giving the samples there the value 0. The output sampling matrix is $V_O = V_I L^{-1}$. The lower part of Fig. 2 shows how the upper left raster is downsampled (lower left), and the upper right raster is upsampled (lower right). Of course, the matrices M and L can be any non-singular integral matrix. Consider again the upper part of Fig. 2. The raster points $n = Vk,\ k \in \{\mathscr{P} \cap \mathbb{Z}^2\}$ have a support given by the integral points in a *support tile* defined by the columns of the support matrix $Q = V^{-1}P$.

$$Q = \begin{pmatrix} 3.5 & 0 \\ 0 & 2 \end{pmatrix}$$

for the sample raster at the left, and

[1] In many practical applications $n \in \mathbb{Z}^m$.

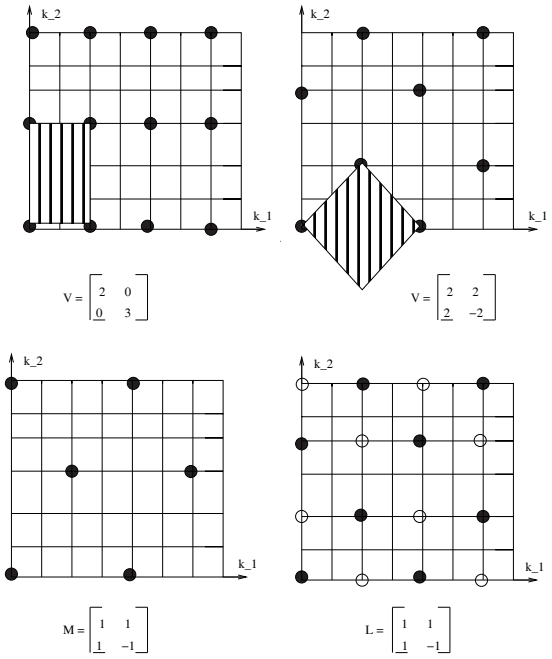


Fig. 2 Rectangular and hexagonal sampling (upper), and downsampling and upsampling (lower)

$$Q = \begin{pmatrix} 1.75 & 1.5 \\ 1.75 & -1.5 \end{pmatrix}$$

for the sample raster at the right.

If W_I is the support matrix at the input of an MD downsampler or upsampler, then the support matrix W_O at the output is $W_O = M^{-1}W_I$ for the downsampler, and $W_O = LW_I$ for the upsampler. In the example shown in Fig. 2, the output support matrix for the downsampling example is $M^{-1}V^{-1}P$, and for the upsampling example it is $LV^{-1}P$. They are

$$W_O = \begin{pmatrix} 1.75 & 1.0 \\ 1.75 & -1.0 \end{pmatrix}$$

for the downsampler (bottom left) in Fig. 2, and

$$W_O = \begin{pmatrix} 3.5 & 0 \\ 0 & 3 \end{pmatrix}$$

for the upsampler (bottom right) in Fig. 2. The characterization of input-output sampling matrices, and input-output support matrices is not enough to set up *balance equations* because it still remains to be determined how the points in the support tiles are ordered. For example, consider a source actor S with sampling matrix $V_O = I$ (the identity matrix), and support matrix $W_O = diag(3,3)$ (producing three rows and three columns, thus having a sampling rate (3, 3)) connected to an upsampler with

input sampling matrix $V_I = I$. The upsampler can scan its input in row scan order. Its output support matrix W_O is $W_O = LW_I$. Thus the upsampler has to output $|\ det(L)\ |$ samples in a parallelepiped $\mathscr{L}$ defined by the columns of the matrix L for every input sample. For example, let

$$L = \begin{pmatrix} 2 & 2 \\ 3 & -2 \end{pmatrix}$$

The first column of L can be interpreted as the upsampler's *horizontal direction*, and the second column as its *vertical direction* in the *generalized rectangle* $\mathscr{L}$. There are L_1= 5 groups of L_2 = 2 samples (that is $L_1 \times L_2 =|\ det(L)\ |$ = 10 samples) in the generalized rectangle. The sampling rate at the output of the upsampler can be chosen to be (L_2, L_1), that is L_2 rows and L_1 columns in natural order. The relation between the natural ordered samples and the samples in the generalized rectangle $\mathscr{L}$ is then

$$\begin{pmatrix} (0,0)\ (1,0)\ (2,0)\ (3,0)\ (4,0)\ \ (0,1)\ \ \ (1,1)\ \ \ (2,1)\ \ (3,1)\ (4,1) \\ (0,0)\ (0,1)\ (0,2)\ (1,2)\ (1,3)\ (-1,1)\ (-1,2)\ (-1,3)\ (0,3)\ (0,4) \end{pmatrix}$$

This is shown in Fig. 3.

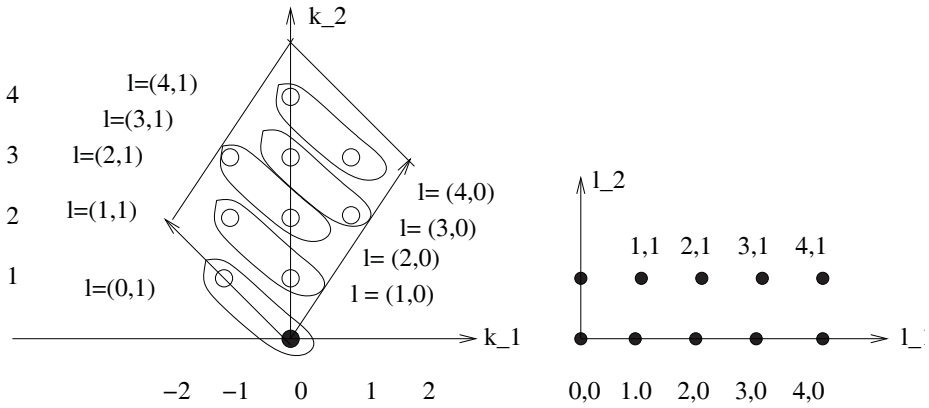


Fig. 3 Ordering of the upsampler's output tokens

In this figure, $\mathscr{L}$ is shown at the left, and the natural ordering of samples in it is shown at the right. The correspondence between samples in $\mathscr{L}$ and the natural ordering of these samples is displayed at the left as well.

1.3 A complete example

Consider the MDSDF graph shown in Figure 4.

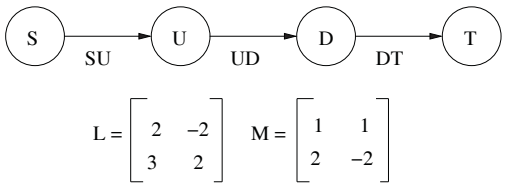


Fig. 4 A combined upsampling and downsampling example

It consists of a chain of four actors S, U, D, and T. S and T are source and sink actors, respectively. U is an expander

$$L = \begin{pmatrix} 2 & -2 \\ 3 & 2 \end{pmatrix}$$

and D is a compressor

$$M = \begin{pmatrix} 1 & 1 \\ 2 & -2 \end{pmatrix}$$

The arcs are labeled SU, UD, and DT, respectively. Let V_{SU} be the identity matrix, and W_{SU}=diag(3, 3). The expander consumes the samples from the source in row scan order, The output samples of the upsampler are ordered in the way discussed above, one at a time (that is as (1,1)), releasing $\mid det(L) \mid$ samples at the output on each invocation. The decimator can consume the upsampler's output samples in a rectangle of samples defined by a factorization $M_1 \times M_2$ of $\mid det(M) \mid$. This rectangle is deduced from the ordering given in Figure 3. Thus, for a 2×2 factorization, the (0,0) invocation of the downsampler consumes the (original) samples (0,0), (-1,1), (0,1) and (-1,2) or equivalently, the natural ordered samples (0,0), (0,1), (1,0), and (1,1).

Of course, other factorizations of $\mid det(M) \mid$ are possible, and the question is whether a factorization can be found for which the total number of samples output by the compressor equals the total number of samples consumed divided by $\mid det(M) \mid$ in a complete cycle as determined by the repetitions matrix which, in turn, depends on the factorizations of $\mid det(L) \mid$ and $\mid det(M) \mid$. Thus, the question is whether $N(W_{DT}) = \frac{N(W_{UD})}{|det(M)|}$. It can be shown [18] that this is not the case for the 2×2 factorization of $\mid det(M) \mid$. The condition is satisfied for a 1×4 factorization, though. Thus, denoting by $r_{X,1}$ and $r_{X,2}$ the repetitions of a node X in the *horizontal* direction and the *vertical* direction, respectively, the balance equations in this case become

$$\begin{pmatrix} 3 \times r_{S,1} = 1 \times r_{U,1} \\ 3 \times r_{S,2} = 1 \times r_{U,2} \\ 5 \times r_{U,1} = 1 \times r_{D,1} \\ 2 \times r_{U,2} = 4 \times r_{D,2} \\ r_{D,1} = r_{T,1} \\ r_{D,2} = r_{T,2} \end{pmatrix}$$

where it is assumed that the decimator produces (1,1) tokens at each firing, and the sink actor consumes (1,1) tokens at each firing.

The resulting elements in the repetitions matrix are

$$\begin{pmatrix} r_{S,1} = 1 & r_{S,2} = 2 \\ r_{U,1} = 3 & r_{U,2} = 6 \\ r_{D,1} = 15 & r_{D,2} = 3 \\ r_{T,1} 15 & r_{T,2} = 3 \end{pmatrix}$$

Now, for a complete cycle we have,

$W_{SU} = diag(3,3) \times diag(r_{S,1}, r_{S,2})$, $W_{UD} = LW_{SU}$, and $W_{DT} = M^{-1}W_{UD}$.

Because W_{UD} is an integral matrix, $N(W_{UD}) = \mid det(W_{UD)} \mid = 180$. There is no simple way to determine $N(W_{DT})$ because this matrix is not an integral matrix. Nevertheless $N(W_{DT})$ can be found to be 45, which is, indeed, a quarter of $N(W_{UD}) = 180$. The reader may consult [18] for further details.

1.4 Initial tokens on arcs

In the example in the previous subsection, the arcs where void of initial tokens. In unidimensional SDF, initial tokens on an arc are a prefix to the stream of tokens output by the arc's source actor. Thus, if an arc's buffer contains two initial tokens, then the first token output by the arc's source node is actually the third token in the sequence of tokens to be consumed by the arc's destination actor. In multidimensional SDF this "shift" is a translation along the vectors of a support matrix W (in the generalized rectangle space) or along the vectors of a sampling matrix V.

Thus for a source node with output sampling matrix

$$V = \begin{pmatrix} 2 & 0 \\ 1 & 1 \end{pmatrix}$$

a multidimensional delay on its outgoing arcs of (1,2) means that the first token released on this arc is actually located at (2,2) in the sampling space.

Similarly, for a support matrix

$$W = \begin{pmatrix} 1 & 0 \\ -2 & 4 \end{pmatrix}$$

a delay of (1,2) means that the first token output is located at (1,2) in the generalized rectangle's natural order. See [18] for further details.

2 Windowed synchronous/cyclo-static dataflow

As discussed in Section 1, specifying applications that process multidimensional arrays in terms of one-dimensional models of computation leads to cumbersome representations. Modeling multidimensional streaming applications as CSDF graphs, for instance, almost always leads to CSDF actors having many phases in their vector valued token production and consumption cycle which incur high control overhead (See Chapter 30). As was shown in Section 1, even the simplest of all dataflow graph models is expressive enough to deal with many multidimensional streaming applications if only tokens of dimension higher than one are introduced in the model.

Unfortunately, there is a large class of image and video signal processing applications, among which are wavelet transforms, noise filtering, edge detection, and medical imaging, for which *sliding window* algorithms have been developed that can achieve higher execution throughput as compared with non-windowed versions.

In windowed algorithms, a computation of an output token typically requires a set of input tokens that are grouped in a *window* of tokens, and windows do overlap when successive output tokens are computed. These windows are multidimensional when tokens in a stream of tokens are multidimensional. Moreover, in most of the sliding image and video application algorithms, the borders of the images need special treatment without explicitly introducing nasty cyclostationary behavior. In particular, pixel planes that are produced by a source actor have to be extended with borders to ensure that pixel plane sizes at the input and the output of pixel plane transforming actors are consistent.

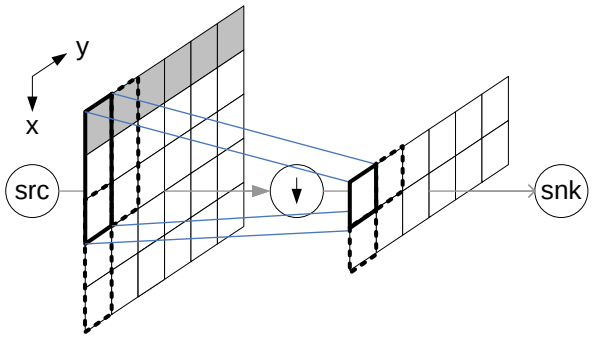


Fig. 5 Example sliding window algorithm showing a source actor (src), a vertical downsampler, and a sink (snk). Gray shaded pixels are extended border pixels.

Fig. 5 shows a simple example. A source actor produces a 4×6 pixel plane that is to be downsampled in row dimension using a sliding window. The gray shaded pixels are border pixels that are not part of the source of the pixel plane produced by the source, but are added to ensure a correct size of the output pixel plane. The window spans three rows in a single column. The downsampler produces one pixel per window, and the window slides two pixel down in row dimension. The windows do not overlap when moving from one column to the next one. Because two consecutive windows in row dimension have two tokens in common, reading of tokens in a window is not necessarily destructive.

The occurrence of overlapping windows, and *virtual border* processing prevent that kind of processing to be represented as an MDSDF graph of the type introduced in Section 1. To overcome this omission, the MDSDF model has been extended to include SDF/CSDF representations of windowed multidimensional streaming applications [12]. The extended model is called the *Windowed Static Data Flow* graph model (WSDF).

In the remainder of this section it is assumed that sampling matrices are diagonal matrices, and that pixel planes are scanned in raster scan order.

2.1 Windowed synchronous/cyclo-static data flow graphs

A WSDF graph consists of nodes — or actors — and edges, as do all dataflow graphs. However, a WSDF edge is to be annotated with more parameters than is the case for SDF, CSDF or MDSDF, because of its extended expressiveness.

Fig. 6 depicts a simple WSDF graph consisting of a *source* node A, a *sink* node B, and an edge (A, B) from source node to sink node.

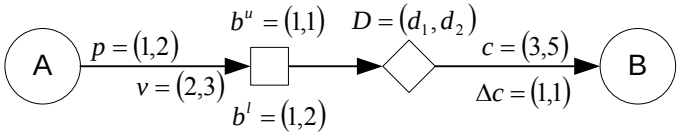


Fig. 6 A simple WSDF graph with two actors and one edge

The source node produces p=(1, 2) tokens (one row, and two columns of a two-dimensional array, that is) per invocation, which is similar to what a source node in a MDSDF graph would do. [2] The token consumption rule for the sink node in WSDF, however, is slightly more involved than in its MDSDF counterpart because windowing and border processing have to be taken into account. The sink node does not consume c=(3,5) tokens, but it has access to tokens in a 3×5 window of tokens.

[2] Recall that only one of the dimensions can be the *streaming* dimension. This could be the row-dimension for the current example.

The movement of that window is governed by the edge parameter Δc, which is (1,1) in the example in Fig. 6, and would have been (2, 1) in the example in Fig. 5. The first window is located in the upper left part of the extended array, and further window positions are determined by the window movement parameter Δc.

WSDF edges can carry initial tokens in exactly the same way as MDSDF edges do. In Fig. 6, there are D=(d_1, d_2) initial tokens, or equivalently, there is a delay D.

The parameters b^u=(b_1^u,b_2^u) = (1, 1) and b^l=(b_1^l,b_2^l) = (1, 2) above and below the edge's border processing rectangle, respectively, in Fig. 6 indicate that there is a border extension of b_1^u rows at the upper side, and b_1^l rows at the lower side of a (rectangular) group of array elements, as well as b_2^u columns at the left side, and b_2^l columns at the right side. Negative values of b_y^x indicate that array entries are suppressed.

The group of array elements that is to be border-extended defines a *virtual token* v. The virtual token in Fig. 6 is v=(2, 3). In contrast with the tokens p=(1, 2) that are effectively produced by the source node A, the virtual token v=(2, 3) is not produced as such, but only specifies the array of elements that is to be border-extended. To distinguish between p and v, the token p is called an *effective* token, and the token v is called a *virtual token*.

Fig. 7 and Fig. 8 illustrate the virtual token and bordering concepts.

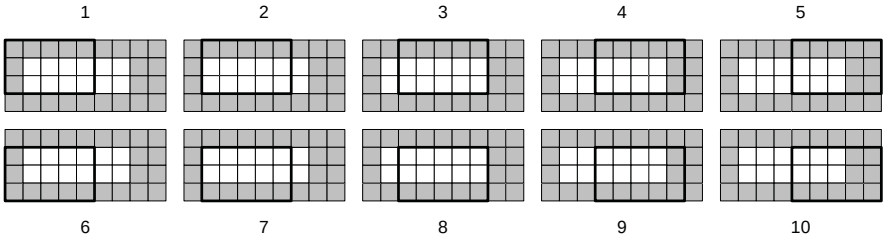


Fig. 7 Different variants of border extension, and the resulting sliding window positions: single virtual token.

Fig. 8, corresponds to the example in Fig. 6, where the virtual token is v=(2, 3). The source node produces an effective token p=(1,2) out of a 2×6 array. The virtual token v=(2, 3) specifies that this array is to be partitioned into two 2×3 subarrays (or *units*), each one being border-extended independently according to b^u=(1, 1), and b^l=(1, 2). The sliding of the window c=(3,5) as determined by the parameter Δc applies to each unit independently. Thus, in Fig. 8, within a single unit, the 3×5 window moves by one column to the right, and after returning to the left border, one row down, and again one column to the right. And similar for the second unit.

Fig. 7 shows the sliding window operation for the case that the virtual token is v=(2,6). In this case, all array elements produced by actor A are combined to a single unit that is border-extended. Consequently, 2×5 different sliding windows are necessary in order to cover the complete array.

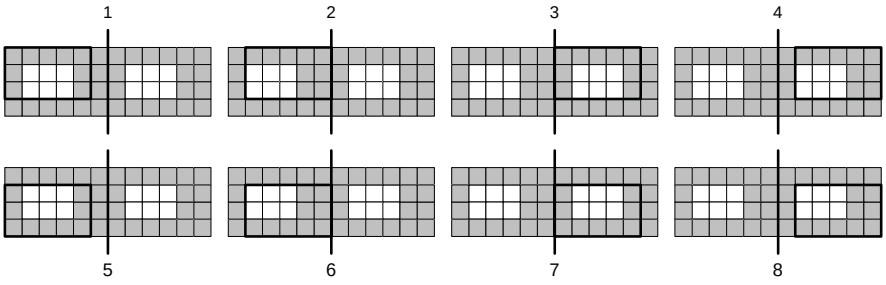


Fig. 8 Different variants of border extension, and the resulting sliding window positions: array split into two units, separated by a vertical dash.

2.1.1 Determination of extended border values

Extended border array elements are not produced by the source actor. They are added and given values by the *border processing* part of the underlying algorithm. Typically, border element values are derived from element values in the source-produced effective token array and other border elements.

Border processing does not effect token production and consumption in the WSDF model. For a producer/consumer pair, the producer produces effective tokens, and the consumer has access to a window of array elements which may contain extended border elements. However, when border element values have to be computed from source-produced elements and other border elements, the elements to be computed as well as the used elements must be contained in the window.

2.2 Balance equations

As is the case with SDF, CSDF, and MDSDF, WSDF has a sample rate consistent periodic behavior and executes in bounded memory. Thus, given a WSDF graph, its initial state, i.e., the initial tokens on its arcs, and the initial position of the sliding windows, the graph can transform token sequences indefinitely in a periodic fashion such that, in each period, the actors are invoked a constant number of times, and at the end of each period the graph returns to its initial state, and sliding windows are back in their initial positions. In other words, a WSDF graph obeys *balance equations* in the same way as SDF, CSDF, and MDSDF graphs do.

Because of the application of a sliding window in WSDF, successive invocations of an actor within a virtual token consume different numbers of array elements. As a consequence, WSDF balance equations resemble CSDF balance equations (see Chapter 30), one equation per dimension.

The first step in the proof is to establish the minimum number of times actors have to be invoked in each dimension to ensure that complete virtual tokens are consumed. For a single edge e, this amount of invocations w_e can be obtained by

calculating how many sliding windows fit into the corresponding virtual token surrounded by extended borders:

$$w_e = \frac{v_e + b_e^u + b_e^l - c_e}{\Delta c_e} + 1 \tag{1}$$

In this equation, v_e, b_e^u, b_e^l, c_e, and Δc_e are the edge's virtual token size, upper and lower boundary extensions, the window size, and window step size, respectively, in terms of number of rows for the row dimension, and of columns in the column dimension. Thus there is an equation $w_{e,x}$ for every dimension x. For the example in Fig. 6, this would lead to

$$\begin{pmatrix} w_{e,1} = \frac{2+1+1-3}{1} + 1 = 2 \\ w_{e,2} = \frac{3+2+1-5}{1} + 1 = 2 \end{pmatrix}$$

Clearly, the sink actor B in Fig. 6 has to be invoked twice in both row and column dimensions to consume a complete (border-extended) virtual token. See also Fig. 8.

In the case that an actor A has several input edges, the minimum number of invocations N_A of actor A equals the smallest common multiple of the minimum number of invocations w_{e_k} among this set of input edges. If there are no input edges, N_A is set to 1.

Now the balance equations ensure, for each edge e, that tokens produced by the source node have been consumed by the sink node, and that both the source node and the sink node have exchanged complete virtual tokens to return to the initial state. The structure of such a balance equation for an edge e is as follows:

$$N_{src(e)} \cdot p_e \cdot q_{src(e)} = \frac{N_{snk(e)}}{w_e} \cdot v_e \cdot q_{snk(e)} \tag{2}$$

where p_e and v_e are the effective and virtual token parameters of edge e for a single dimension. q_x defines the number of times the minimum period of actor x is to be executed. This number follows from the basic balance equation

$$\Gamma \cdot q = 0. \tag{3}$$

The topology matrix Γ has one column for each actor, and one row for each edge. There is one basic balance equation for each dimension. The entries of the matrix Γ are $N_A \times p_A$ when actor A is a source actor, $-\frac{N_A}{w_A} \times v_A$ when actor A is a sink actor, and 0 otherwise.

For a connected graph, the matrix Γ has to be of rank $\mid V \mid$ -1, where $\mid V \mid$ is the cardinality of the set of actors V. Finally, the repetitions vector r satisfies the equation

$$r = N \cdot q \tag{4}$$

where the matrix N is a diagonal matrix with entries N_{A_k}, $k = 1 \ldots \mid V \mid$.

The set of vectors q, one for each dimension, gives a matrix Q which in two dimensions would be

$$Q = \begin{pmatrix} q_{A_1,1} & \cdots & q_{A_n,1} \\ q_{A_1,2} & \cdots & q_{A_n,2} \end{pmatrix}.$$

Similarly, the set of repetitions vectors r, gives a repetitions matrix R which in two dimensions would be

$$R = \begin{pmatrix} r_{A_1,1} & \cdots & r_{A_n,1} \\ r_{A_1,2} & \cdots & r_{A_n,2} \end{pmatrix}. \tag{5}$$

2.2.1 A complete example

Fig. 9 shows a WSDF graph specification of a medical image enhancement multi-resolution filter application. Actor $S1$ produces a 480×640 image that is down-sampled by actor $C1$. The downsampled image is sent to actor $F2$, as well as to an upsampler $E1$ that returns an image of size 480×640. Actor $F1$ operates on a sliding window c= (3, 3). Actor $A1$ combines the results of actors $F1$ and $F2$. Actor $E2$ ensures that both inputs to actor $A1$ have compatible sizes.

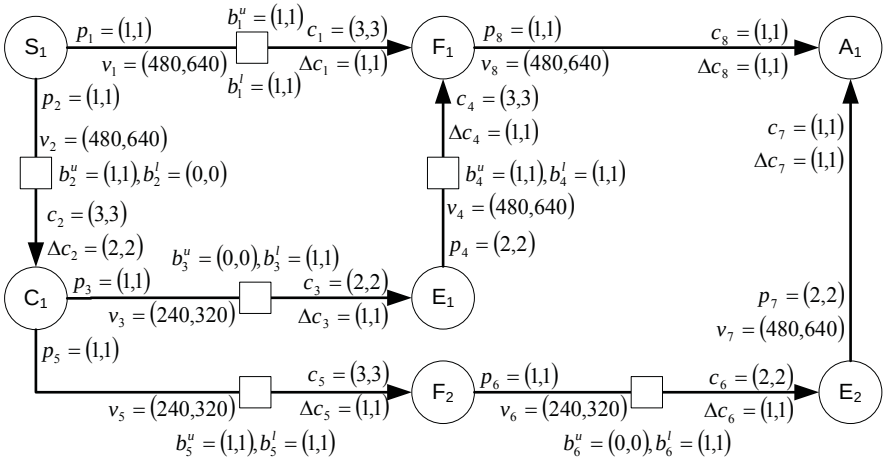


Fig. 9 WSDF graph representation of a multi-resolution filter

Application of equation (1), and calculation of the smallest common multiple for all input edges of actors yields the minimum number of actor invocations needed to consume complete virtual tokens (images in this case). They are listed in the following table.

Actor	$S1$	$F1$	$A1$	$C1$	$E1$	$F2$	$E2$
Row	1	480	480	240	240	240	240
Column	1	640	640	320	320	320	320

Application of equation (2) gives the balance equations

Row	Column
$e1: \quad 1 \cdot 1 \cdot q_{S1,1} = \frac{480}{480} \cdot 480 \cdot q_{F1,1}$	$e1: \quad 1 \cdot 1 \cdot q_{S1,2} = \frac{640}{640} \cdot 640 \cdot q_{F1,2}$
$e2: \quad 1 \cdot 1 \cdot q_{S1,1} = \frac{240}{240} \cdot 480 \cdot q_{C1,1}$	$e2: \quad 1 \cdot 1 \cdot q_{S1,2} = \frac{320}{320} \cdot 640 \cdot q_{C1,2}$
$e3: 240 \cdot 1 \cdot q_{C1,1} = \frac{240}{240} \cdot 240 \cdot q_{E1,1}$	$e3: 320 \cdot 1 \cdot q_{C1,2} = \frac{320}{320} \cdot 320 \cdot q_{E1,2}$
$e4: 240 \cdot 2 \cdot q_{E1,1} = \frac{480}{480} \cdot 480 \cdot q_{F1,1}$	$e4: 320 \cdot 2 \cdot q_{E1,2} = \frac{640}{640} \cdot 640 \cdot q_{F1,2}$
$e5: 240 \cdot 1 \cdot q_{C1,1} = \frac{240}{240} \cdot 240 \cdot q_{F2,1}$	$e5: 320 \cdot 1 \cdot q_{C1,2} = \frac{320}{320} \cdot 320 \cdot q_{F2,2}$
$e6: 240 \cdot 1 \cdot q_{F2,1} = \frac{240}{240} \cdot 240 \cdot q_{E2,1}$	$e6: 320 \cdot 1 \cdot q_{F2,2} = \frac{320}{320} \cdot 320 \cdot q_{E2,2}$
$e7: 240 \cdot 2 \cdot q_{E2,1} = \frac{480}{480} \cdot 480 \cdot q_{A1,1}$	$e7: 320 \cdot 2 \cdot q_{E2,2} = \frac{640}{640} \cdot 640 \cdot q_{A1,2}$
$e8: 480 \cdot 1 \cdot q_{F1,1} = \frac{480}{480} \cdot 480 \cdot q_{A1,1}$	$e8: 640 \cdot 1 \cdot q_{F1,2} = \frac{640}{640} \cdot 640 \cdot q_{A1,2}$

The corresponding matrices have rank $\mid V \mid -1 = 7 - 1 = 6$. Consequently, a minimal integer solution exists as shown in the following table.

Actor	$S1$	$F1$	$A1$	$C1$	$E1$	$F2$	$E2$
Row	480	1	1	1	1	1	1
Column	640	1	1	1	1	1	1

Finally equation (5) leads to the repetitions given in the following table.

Actor	$S1$	$F1$	$A1$	$C1$	$E1$	$F2$	$E2$
Row	480	480	480	240	240	240	240
Column	640	640	640	320	320	320	320

This confirms that the WSDF specification is correct in that it does not cause unbounded accumulation of tokens. In addition, it demonstrates that the WSDF graph works as expected in that actors $E1$, $F2$, and $E2$ execute less often than the other actors since they read small images at their input. Furthermore, also actor $C1$ is less active, since it performs a downsampling operation. To this end, the sliding window moves by two pixels while each actor invocation only generates one pixel.

3 Motivation for dynamic DSP-oriented dataflow models

The decidable dataflow models covered in Chapter 30 are useful for their predictability, strong formal properties, and amenability to powerful optimization techniques. However, for many signal processing applications, it is not possible to represent all of the functionality in terms of purely decidable dataflow representations. For example, functionality that involves conditional execution of dataflow subsystems or actors with dynamically varying production and consumption rates generally cannot be expressed in decidable dataflow models.

The need for expressive power beyond that provided by decidable dataflow techniques is becoming increasingly important in design and implementation signal processing systems. This is due to the increasing levels of application dynamics that

must be supported in such systems, such as the need to support multi-standard and other forms of multi-mode signal processing operation; variable data rate processing; and complex forms of adaptive signal processing behaviors.

Intuitively, *dynamic dataflow* models can be viewed as dataflow modeling techniques in which the production and consumption rates of actors can vary in ways that are not entirely predictable at compile time. It is possible to define dynamic dataflow modeling formats that are decidable. For example, by restricting the types of dynamic dataflow actors, and by restricting the usage of such actors to a small set of graph patterns or "schemas", Gao, Govindarajan, and Panangaden defined the class of *well-behaved dataflow graphs*, which provides a dynamic dataflow modeling environment that is amenable to compile-time bounded memory verification [25].

However, most existing DSP-oriented dynamic dataflow modeling techniques do not provide decidable dataflow modeling capabilities. In other words, in exchange for the increased modeling flexibility (expressive power) provided by such techniques, one must typically give up guarantees on compile-time buffer underflow (deadlock) and overflow validation. In dynamic dataflow environments, analysis techniques may succeed in guaranteeing avoidance of buffer underflow and overflow for a significant subset of specifications, but, in general, specifications may arise that "break" these analysis techniques — i.e., that result in inconclusive results from compile-time analysis.

Dynamic dataflow techniques can be divided into two general classes: 1) those that are formulated explicitly in terms of interacting combinations of state machines and dataflow graphs, where the dataflow dynamics are represented directly in terms of transitions within one or more underlying state machines; and 2) those where the dataflow dynamics are represented using alternative means. The separation in this dichotomy can become somewhat blurry for models that have a well-defined state structure governing the dataflow dynamics, but whose design interface does not expose this structure directly to the programmer. Chapter 36 includes coverage of dynamic dataflow techniques in the first category described above — in particular, those based on explicit interactions between dataflow graphs and finite state machines. In this chapter, we focus on the second category. Specifically, we discuss dynamic dataflow modeling techniques that involve different kinds of modeling abstractions, apart from state transitions, as the key mechanisms for capturing dataflow behaviors and their potential for run-time variation.

Numerous dynamic dataflow modeling techniques have evolved over the past couple of decades. A comprehensive coverage of these techniques, even after excluding the "state-centric" ones, is out of the scope this chapter. Instead we provide a representative cross-section of relevant dynamic dataflow techniques, with emphasis on techniques for which useful forms of compile time analysis have been developed. Such techniques can be important for exploiting the specialized properties exposed by these models, and improving predictability and efficiency when deriving simulations or implementations.

4 Boolean dataflow

The Boolean dataflow (BDF) model of computation extends synchronous dataflow with a class of dynamic dataflow actors in which production and consumption rates on actor ports can vary as two-valued functions of *control tokens*, which are consumed from or produced onto designated *control ports* of dynamic dataflow actors. An actor input port is referred to as a *conditional input port* if its consumption rate can vary in such a way, and similarly an output port with a dynamically varying production rate under this model is referred to as a *conditional output port*.

Given a conditional input port p of a BDF actor A, there is a corresponding input port C_p, called the *control input* for p, such that the consumption rate on C_p is statically fixed at one token per invocation of A, and the number of tokens consumed from p during a given invocation of A is a two-valued function of the data value that is consumed from C_p during the same invocation.

The dynamic dataflow behavior for a conditional output port is characterized in a similar way, except that the number of tokens produced on such a port can be a two-valued function of a token that is consumed from a control input port or of a token that is produced onto a *control output* port. If a conditional output port q is controlled by a control output port C_q, then the production rate on the control output is statically fixed at one token per actor invocation, and the number of tokens produced on q during a given invocation of the enclosing actor is a two-valued function of the data value that is produced onto C_q during the same invocation of the actor.

Two fundamental dynamic dataflow actors in BDF are the *switch* and *select* actors, which are illustrated in Figure 10. The switch actor has two input ports, a control input port w_c and a *data* input port w_d, and two output ports w_x and w_y. The port w_c accepts Boolean valued tokens, and the consumption rate on w_d is statically fixed at one token per actor invocation. On a given invocation of a switch actor, the data value consumed from w_d is copied to a token that is produced on either w_x or w_y depending on the Boolean value consumed from w_c. If this Boolean value is `true`, then the value from the data input is routed to w_x, and no token is produced on w_y. Conversely if the control token value is `false`, then the value from w_d is routed to w_y with no token produced on w_x.

A BDF select actor has a single control input port s_c; two additional input ports (*data input ports*) s_x and s_y; and a single output port s_o. Similar to the control port of the switch actor, the s_c port accepts Boolean valued tokens, and the production rate on the s_o port is statically fixed at one token per invocation. On each invocation of the select actor data is copied from a single token from either s_x or s_y to s_o depending on whether the corresponding control token value is `true` or `false` respectively.

Switch and select actors can be integrated along with other actors in various ways to express different kinds of control constructs. For example, Figure 11 illustrates an if-then-else construct, where the actors A and B are applied conditionally based on a stream of control tokens. Here A and B are SDF actors that each consume one token and produce one token on each invocation.

Buck has developed scheduling techniques to automatically derive efficient control structures from BDF graphs under certain conditions [24]. Buck has also shown

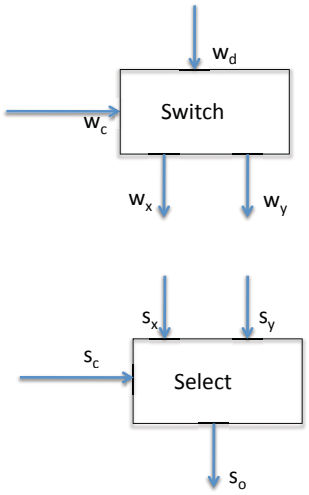


Fig. 10 Switch and select actors in Boolean dataflow.

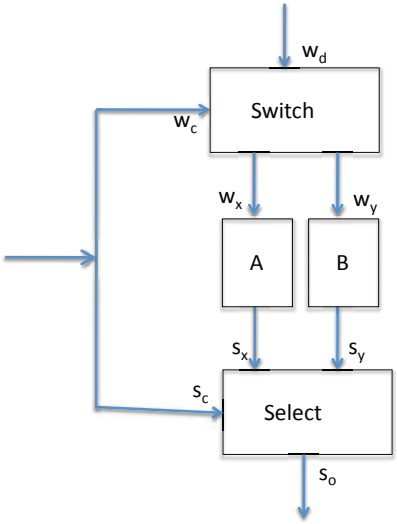


Fig. 11 An if-then-else construct expressed in terms of Boolean dataflow.

that BDF is Turing complete, and furthermore, that SDF augmented with just switch and select (and no other dynamic dataflow actors) is also Turing complete. This latter result provides a convenient framework with which one can demonstrate Turing completeness for other kinds of dynamic dataflow models, such as the enable-invoke dataflow model described in Section 8. In particular, if a given model of computa-

tion can express all SDF actors as well as the functionality associated with the BDF switch and select actors, then such a model can be shown to be Turing complete.

5 The Stream-based Function Model

One of the dynamic dataflow models is the *Stream-based Function model*. In fact, it is a model for the actors in dataflow graphs that otherwise obey in principle the dataflow model of computation communication semantics: Actors communicate point-to-point over unbounded FIFO channels, and an actor blocks until sufficient tokens are available in the currently sequentially addressed channel.

5.1 The concern for an actor model

It is convenient to call the active entities in dataflow graphs and networks *actors*, whether they are void of state, encompass a single thread of control, or are processes. Thus static dataflow graphs (Chapter 30), dynamic dataflow graphs, polyhedral process networks (Chapter 33), and Kahn process networks (Chapter 34) can collectively be referred to as *Dataflow Actor Networks* (DFAN). They obey the Kahn actor coordination semantics, possibly augmented with actor firing rules, annotated with deadlock free minimal FIFO buffer capacities, and having one or more global schedules associated with them. A DFAN is very often an appealing *specification model* for signal processing applications. In many cases, the signal processing application DFAN specification is to be implemented in a single-chip multiprocessor execution platform that consists of a number of computational elements (processors), and a communication, synchronization, and storage infrastructure.

This requires a *mapping* that relates an untimed application model and a timed architecture model together: Actors are assigned to processors, FIFO channels are logically embedded in — often distributed — memory, and DFAN communication primitives and protocols are transformed to platform specific communication primitives and protocols. The set of platform computational components may not be homogeneous, nor may they support multiple threads of control. Thus, the specification of platform processors and DFAN actors may be quite different. In fact, the actors in a DFAN are poorly modeled, as they are specified in a standard sequential host programming language, by convention. And platform processors may be not modeled at all, because they may be dedicated and depending on the specification of the actors that are to be assigned to them. For example, when an actor is a process in which several threads of control can be recognized, it would be difficult, if at all possible, to relate it transparently and time consciously to a computational component that can only support a single thread of control, that has no operating system embedded in it, that is. To overcome this problem, it makes sense to provide an *actor model* that can serve as an intermediate between the conventional DFAN actor

specification, and a variety of computational elements in the execution platform. The *Stream-based Function Model* [16] is exactly aimed at that purpose.

5.2 The stream-based function actor model

The generic actor in a DFAN is a process that comprises one or more code segments consisting of a sequence of control statements, read and write statements, and function-call statements, that are not structured in any particular way. The stream-based function actor model aims at providing a process structure that makes the internal behavior of actors transparent and time conscious. It is composed of a *set of functions*, called *function repertoire*, a *transition and selection function*, called *controller*, and a combined function and data *state*, called *private memory*. The controller selects a function from the function repertoire that is associated with a *current function state*, and makes a transition to the *next function state*. A selected function is *enabled* when all its input arguments can be read, and all its results can be written; It is blocked otherwise. A selected non-blocked function must evaluate or *fire*. Arguments and results are called *input tokens* and *output tokens*, respectively.

An actor operates on sequences of tokens, streams or signals, as a result of a repetitive enabling and firing of functions from the function repertoire. Tokens are *read* from external ports and/or loaded from private memory, and are *written* to external ports and/or stored in private memory. The external ports connect to FIFO channels through which actors communicate point-to-point.

Figure 12 depicts an illustrative stream-based function (SBF) actor. Its function repertoire is $P = \{f_{init}, f_a, f_b\}$. The two read ports, and the single write port connect to two input and one output FIFO channels, respectively. P contains at least an initialization function f_{init}, and is a finite set of functions. The functions in P must be selected in a mutual exclusive order. That order depends on the controller's selection and transition function. The private memory consists of a function state part, and a data state part that do not intersect.

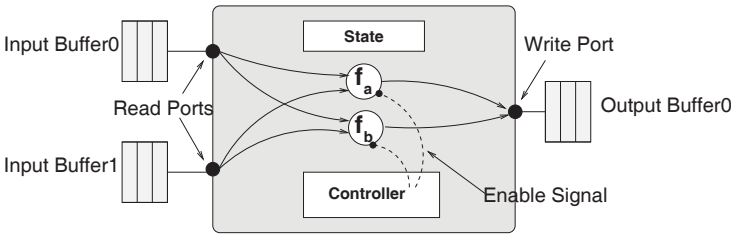


Fig. 12 A simple SBF actor.

5.3 The formal model

If C denotes the SBF actor's function state space, and if D denotes its data state space, then the actor's state space S is the Cartesian product of C and D,

$$S = C \times D,\ C \cap D = \emptyset. \tag{6}$$

Let $c \in C$ be the current function state. From c, the controller selects a function and makes a function state transition by activating its selection function μ and transition function ω as follows,

$$\mu : C \to P,\ \mu(c) = f, \tag{7}$$

$$\omega : C \times D \to C,\ \omega(c,d) = c'. \tag{8}$$

The evaluation of the combined functions μ and ω is instantaneous. The controller cannot change the content of C; it can only observe it. The transition function allows for a dynamic behavior as it involves the data state space D. When the transition function is only a map from C to C, then the trajectory of selected functions will be static or cyclo-static (see Chapter 30 and Chapter 33). Assuming that the DFAN does not deadlock, and that its input streams are not finite, the controller will repeatedly invoke functions from the function repertoire P in an endless firing of functions,

$$f_{init} \xrightarrow{\mu(\omega(S))} f_a \xrightarrow{\mu(\omega(S))} f_b \xrightarrow{\mu(\omega(S))} \ldots f_x \xrightarrow{\mu(\omega(S))} \ldots \tag{9}$$

Such a behavior is called a *Fire-and-Exit* behavior. It is quite different from a threaded approach in that all synchronization points are explicit. The role of the function f_{init} in Eq. 9 is crucial. This function has to provide the starting current function state $c_{init} \in C$. It evaluates first and only once, and it may read tokens from one or more external ports to return the starting current function state.

$$c_{init} = f_{init}(channel_a \ldots channel_z) \tag{10}$$

The SBF model is reminiscent of the *Applicative State Transition* (AST) node in the systolic array model of computation (Chapter 29) introduced by Annevelink [26], after the *Applicative State Transition* programming model proposed by Backus [27]. The AST node, like the SBF actor, comprises a function repertoire, a selection function μ, and a transition function ω. However, the AST node does not have a private memory, and the initial function state c_{init} is read from a unique external channel. Moreover, AST nodes communicate through register channels.

The SBF actor model is also reminiscent of the *California Actor Language Model* (CAL), which is discussed in Chapter 36 and also in section 6 of this chapter.

Clearly, decidable and polyhedral dataflow graphs and networks are special cases of SBF actor networks. Analysis is still possible in case the SBF network models an underlying *weakly dynamic* nested loop program [17]. A sequential nested loop program is said to be weakly dynamic when the loop bounds and the variable in-

dexing functions are affine functions of the loop iterators and static parameters. Conditional statements are without any restriction. Notice that in case conditional statements are also affine in the loop iterators and static parameters, the nested loop program becomes, then, a static affine nested loop program (Chapter 33). Analysis is also possible when, in addition, the parameters are not static [8].

5.4 Communication and scheduling

Actors in dataflow actor networks communicate point to point over FIFO buffered channels. Conceptually, buffer capacities are unlimited, and actors synchronize by means of a *blocking read* protocol: An actor that attempts to read tokens from a specific channel will block whenever that channel is empty. Writing tokens will always proceed. When a function $[c,d] = f(a,b)$ is invoked, it has to bind to input ports p_x and p_y when ports p_x and p_y are to deliver the arguments a and b, in this order. Thus if the n-th channel to which port p_n has access is empty so that the function's n-th argument cannot be delivered, then the function $f(a_1,\ldots,a_n\ldots a_z)$ will block. This is the default DFAN communication synchronization protocol. Of course, channel FIFO capacities are not unlimited in practice, so that mapped DFAN do have a *blocking write* protocol as well: An actor that attempts to write tokens to a specific channel will block whenever that channel is full. Thus a function $[c,d] = f(a,b)$ has to bind to output ports q_x and q_y when ports q_x and q_y are to receive results c and d, in this order. The binding of a function to input and output ports can be *implicit* or *explicit*. In the case of implicit binding, the function is said to know its corresponding input and output ports. Because a particular function may read arguments and write results from non-unique input ports and to non-unique output ports, respectively, the function repertoire will, then, contain *function variants*. Two functions in the function repertoire are function variants when they are functionally identical, and their arguments and results are read from and written to different channels.

In the case of explicit binding, the controller's selection function μ selects both a function from the function repertoire and the corresponding input and output ports.

Explicit binding allows to separate function selection and binding selection, so that reading, executing, and writing can proceed in a pipelined fashion. Both the implicit and explicit binding methods assume blocking read and write actor synchronization semantics. However, the SBF model allows for a more general deterministic dynamic approach. In this approach, the actor behavior is divided into a *channel checking* part and a *scheduling* part. In the channel checking part, channels are visited without empty or full channel blockings. A visited input channel C_{in} returns a $C_{in}.1$ signal when the channel is not empty, and a $C_{in}.0$ signal when the channel is empty. And similarly for output channels. These signals indicate whether or not a particular function from the function repertoire can fire. In the scheduling part, a function can only be invoked when the channel checking signals allow it to

fire. If not, then the function will not be invoked. As a consequence, the actor is blocked. Clearly channel checking and scheduling can proceed in parallel.

5.5 Composition of SBF actors

It should be clear that an SBF actor has a single thread of control. However, when mapping a DFAN onto an execution platform, one may assign two or more actors to a single SBF processing unit. That processing unit may have an SBF-like model, so that a many-to-one mapping must conform to the single thread of control model. This implies that one or more SBF actors in a DFAN have to be merged together in a single SBF actor. An example is shown in Fig. 13.

Part of a DFAN is shown at the left in the figure. The functionality of the actors *A*, *B*, and *C* is also shown. The functions $f_1()$ and $f_2()$ of actor *A* are invoked in sequence, and so are the functions $g_1()$ and $g_2()$ of actor *B*, and the functions $h_1()$ and $h_2()$ of actor *C*. The three actors are merged into a single actor *D* as shown at the right of the figure. When modeling actor *D* as an SBF actor, the function repertoire will contain functions $f_1()$, $f_2()$, $g_1()$, $g_2()$, $h_1()$, and $h_2()$. A possible single thread schedule is also shown at the right in the figure, where $\{PM\}$ stands for *private memory*. However, because actors in the original DFAN are concurrent threads, and because in the SBF actor channel checking and function scheduling proceed in parallel, the actual schedule of functions in the function repertoire of the SBF actor *D* may be different from the one shown. Indeed, if — for example — function $h_1()$ cannot fire, then function $g_1()$ may be fireable and get invoked before function $h_1()$. Although this is a sort of multi-threading, it is safe and deterministic, because there is no preemption, and no context switch.

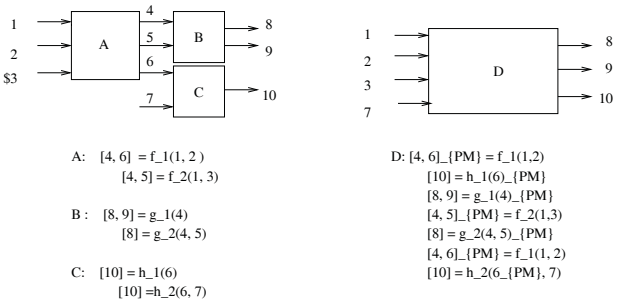


Fig. 13 Merging SBF Actors.

6 CAL

In addition to providing a dynamic dataflow model of computation that is suitable for signal processing system design, CAL provides a complete programming language and is supported by a growing family of development tools for hardware and software implementation. The name "CAL" is derived as a self-referential acronym for the *CAL actor language*. CAL was developed by Eker and Janneck at U. C. Berkeley [15], and has since evolved into an actively-developed, widely-investigated language for design and implementation of embedded software and field programmable gate array applications (e.g., see [10] [3] [19]).

One of the most notable developments to date in the evolution of CAL has been its adoption as part of the recent MPEG standard for reconfigurable video coding (RVC) [1]. Specifically, to enable higher levels of flexibility and adaptivity in the evolution of RVC standards, CAL is used as a format for transmitting the appropriate decoding *functionality* along with the raw image data that comprises a video stream. In other words, CAL programs are embedded along with the desired video content to form an RVC bit stream. At the decoder, the dataflow graph for the decoder is extracted from the bit incoming bit stream, and references to CAL actors are mapped to platform specific library components that execute specific decoding functions. These capabilities for encoding video decoding algorithms as coarse-grain dataflow graphs and encoding specific decoding functions as references to pre-defined library elements allow for highly compact and flexible representation of video decoding systems.

A CAL program is specified as a network of CAL actors, where each actor is a dataflow component that is expressed in terms of a general underlying form of dataflow. This general form of dataflow admits both static and dynamic behaviors, and even non-deterministic behaviors.

Like typical actors in any dataflow programming environment, a CAL actor in general has a set of input ports and a set of output ports that define interfaces to the enclosing dataflow graph. A CAL actor also encapsulates its own private state, which can be modified by the actor as it executes but cannot be modified directly by other actors.

The functional specification of a CAL actor is decomposed into a set of *actions*, where each action can be viewed as a template for a specific class of firings or invocations of the actor. Each firing of an actor corresponds to a specific action and executes based on the code that is associated with that action. The core functionality of actors therefore is embedded within the code of the actions. Actions can in general consume tokens from actor input ports, produce tokens on output ports, modify the actor state, and perform computation in terms of the actor state and the data obtained from any consumed tokens.

The number of tokens produced and consumed by each action with respect to each actor output and input port, respectively, is declared up front as part of the declaration of the action. An action need not consume data from all input ports nor must it produce data on all output ports, but ports with which the action exchanges data, and the associated rates of production and consumption must be constant for

the action. Across different actions, however, there is no restriction of uniformity in production and consumption rates, and this flexibility enables the modeling of dynamic dataflow in CAL.

A CAL actor A can be represented as a sequence of four elements

$$\sigma_0(A), \Sigma(A), \Gamma(A), pri(A), \tag{11}$$

where $\Sigma(A)$ represents the set of all possible values that the state of A can take on; $\sigma_0(A) \in \Sigma(A)$ represents the initial state of the actor, before any actor in the enclosing dataflow graph has started execution; $\Gamma(A)$ represents the set of actions of A; and $pri(A)$ is a partial order relation, called the *priority relation* of A, on $\Gamma(A)$ that specifies relative priorities between actions.

Actions execute based on associated *guard conditions* as well as the priority relation of the enclosing actor. More specifically, each actor has an associated guard condition, which can be viewed as a Boolean expression in terms of the values of actor input tokens and actor state. An actor A can execute whenever its associated guard condition is satisfied (*true*-valued), and no higher-priority action (based on the priority relation $pri(A)$) has a guard condition that is also satisfied.

In summary, CAL is a language for describing dataflow actors in terms of ports, actions (firing templates), guards, priorities, and state. This finer, *intra-actor* granularity of formal modeling within CAL allows for novel forms of automated analysis for extracting restricted forms of dataflow structure. Such restricted forms of structure can be exploited with specialized techniques for verification or synthesis to derive more predictable or efficient implementations.

An example of this capability for *specialized region detection* in CAL programs is the technique of deriving and exploiting so-called *statically schedulable regions* (SSRs) [3]. Intuitively, an SSR is a collection of CAL actions and ports that can be scheduled and optimized statically using the full power of static dataflow techniques, such as those available for SDF, and integrated into the schedule for the overall CAL program through a top-level dynamic scheduling interface.

SSRs can be derived through a series of transformations that are applied on intermediate graph representations. These representations capture detailed relationships among actor ports and actions, and provide a framework for effective quasi-static scheduling of CAL-based dynamic dataflow representations. Here, by *quasi-static scheduling*, we mean the construction of dataflow graph schedules in which a significant proportion of overall schedule structure is fixed at compile-time. Quasi-static scheduling has the potential to significantly improve predictability, reduce run-time scheduling overhead, and as discussed above, expose subsystems whose internal schedules can be generated using purely static dataflow scheduling techniques.

Further discussion about CAL can be found in Chapter 3, which discusses the application of CAL to reconfigurable video coding.

7 Parameterized dataflow

Parameterized dataflow is a meta-modeling approach for integrating dynamic parameters and run-time adaptation of parameters in a structured way into a certain class of dataflow models of computations, in particular, those models that have a well-defined concept of a graph *iteration* [20]. For example, SDF and CSDF, which are discussed in Chapter 30, and MDSDF, which is discussed in Section 1, have well defined concepts of iterations based on solutions to the associated forms of balance equations. Each of these models can be integrated with parameterized dataflow to provide a dynamically parameterizable form of the original model.

When parameterized dataflow is applied in this way to generalize a specialized dataflow model such as SDF, CSDF, or MDSDF, the specialized model is referred to as the *base model*, and the resulting, dynamically parameterizable form of the base model is referred to as *parameterized XYZ*, where *XYZ* is the name of the base model. For example, when parameterized dataflow is applied to SDF as the base model, the resulting model of computation, called *parameterized synchronous dataflow* (*PSDF*), is significantly more flexible than SDF as it allows arbitrary parameters of SDF graphs to be modified at run-time. Furthermore, PSDF provides a useful framework for quasi-static scheduling, where fixed-iteration looped schedules, such as single appearance schedules [23], for SDF graphs can be replaced by *parameterized looped schedules* [20] [11] in which loop iteration counts are represented as symbolic expressions in terms of variables whose values can be adapted dynamically through computations that are derived from the enclosing PSDF specification.

Intuitively, parameterized dataflow allows arbitrary attributes of a dataflow graph to be parameterized, with each parameter characterized by an associated domain of admissible values that the parameter can take on at any given time. Graph attributes that can be parameterized include scalar or vector attributes of individual actors, such as the coefficients of a finite impulse response filter or the block size associated with an FFT; edge attributes, such as the delay of an edge or the data type associated with tokens that are transferred across the edge; and graph attributes, such as those related to numeric precision, which may be passed down to selected subsets of actors and edges within the given graph.

The parameterized dataflow representation of a computation involves three cooperating dataflow graphs, which are referred to as the *body* graph, the *subinit* graph, and the *init* graph. The body graph typically represents the functional "core" of the overall computation, while the subinit and init graphs are dedicated to managing the parameters of the body graph. In particular, each output port of the subinit graph is associated with a body graph parameter such that data values produced at the output port are propagated as new parameter values of the associated parameter. Similarly, output ports of the init graph are associated with parameter values in the subinit and body graphs.

Changes to body graph parameters, which occur based on new parameter values computed by the init and subinit graphs, cannot occur at arbitrary points in time. Instead, once the body graph begins execution it continues uninterrupted through

a graph iteration, where the specific notion of an iteration in this context can be specified by the user in an application-specific way. For example, in PSDF, the most natural, general definition for a body graph iteration would be a single *SDF iteration* of the body graph, as defined by the SDF repetitions vector. Chapter 30 describes in detail this concept of an SDF iteration and the associated concept of the SDF repetitions vector.

However, an iteration of the body graph can also be defined as some constant number of iterations, for example, the number of iterations required to process a fixed-size block of input data samples. Furthermore, parameters that define the body graph iteration can be used to parameterize the body graph or the enclosing PSDF specification at higher levels of the model hierarchy, and in this way, the processing that is defined by a graph iteration can itself be dynamically adapted as the application executes. For example, the duration (or block length) for fixed-parameter processing may be based on the size of a related sequence of contiguous network packets, where the sequence size determines the extent of the associated graph iteration.

Body graph iterations can even be defined to correspond to individual actor invocations. This can be achieved by defining an individual actor as the body graph of a parameterized dataflow specification, or by simply defining the notion of iteration for an arbitrary body graph to correspond to the *next actor firing* in the graph execution. Thus, when modeling applications with parameterized dataflow, designers have significant flexibility to control the windows of execution that define the boundaries at which graph parameters can be changed.

A combination of cooperating body, init, and subinit graphs is referred to as a *PSDF specification*. PSDF specifications can be abstracted as PSDF actors in higher level PSDF graphs, and in this way, PSDF specifications can be integrated hierarchically.

Figure 14 illustrates a PSDF specification for a speech compression system. This illustration is adapted from [20]. Here *setSp* (“set speech”) is an actor that reads a header packet from a stream of speech data, and configures L, which is a parameter that represents the length of the next speech instance to process. The *s1* and *s2* actors are input interfaces that inject successive samples of the current speech instance into the dataflow graph. The actor *s2* zero-pads each speech instance to a length R ($R \geq L$) so that the resulting length is divisible by N, which is the speech segment size. The *An* (“analyze”) actor performs linear prediction on speech segments, and produces corresponding auto-regressive (AR) coefficients (in blocks of M samples), and residual error signals (in blocks of N samples) on its output edges. The actors *q1* and *q2* represent quantizers, and complete the modeling of the transmitter component of the body graph.

Receiver side functionality is then modeled in the body graph starting with the actors *d1* and *d2*, which represent dequantizers. The actor *Sn* (“synthesize”) then reconstructs speech instances using corresponding blocks of AR coefficients and error signals. The actor *Pl* (“play”) represents an output interface for playing or storing the resulting speech instances.

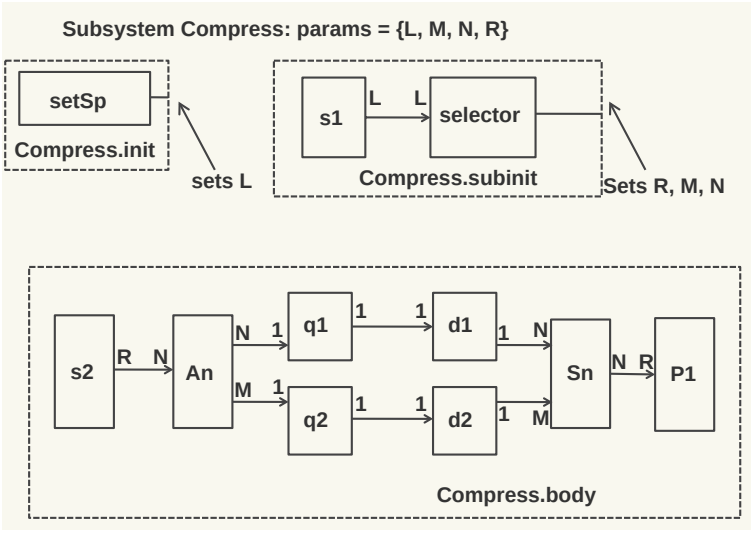


Fig. 14 An illustration of a speech compression system that is modeled using PSDF semantics. This illustration is adapted from [20].

The model order (number of AR coefficients) M, speech segment size N, and zero-padded speech segment length R are determined on a per-segment basis by the *selector* actor in the subinit graph. Existing techniques, such as the Burg segment size selection algorithm and AIC order selection criterion [22] can be used for this purpose.

The model of Figure 14 can be optimized to eliminate the zero padding overhead (modeled by the parameter R). This optimization can be performed by converting the design to a parameterized cyclo-static dataflow (PCSDF) representation. In PCSDF, the parameterized dataflow meta model is integrated with CSDF as the base model instead of SDF.

For further details on this speech compression application and its representations in PSDF and PCSDF, the semantics of parameterized dataflow and PSDF, and quasi-static scheduling techniques for PSDF, we refer the reader to [20].

Parameterized cyclo-static dataflow (PCSDF), the integration of parameterized dataflow meta-modeling with cyclo-static dataflow, is explored further in [13]. The exploration of different models of computation, including PSDF and PCSDF, for the modeling of software defined radio systems is explored in [6]. In [2], Kee et al. explore the application of PSDF techniques to field programmable gate array implementation of the physical layer for 3GPP-Long Term Evolution (LTE). The integration of concepts related to parameterized dataflow in language extensions for embedded streaming systems is explored in [7]. General techniques for analysis and verification of hierarchically reconfigurable dataflow graphs are explored in [14].

8 Enable-invoke dataflow

Enable-invoke dataflow (EIDF) is another DSP-oriented dynamic dataflow modeling technique [9]. The utility of EIDF has been demonstrated in the context of behavioral simulation, FPGA implementation, and prototyping of different scheduling strategies [9] [4] [5]. This latter capability — prototyping of scheduling strategies — is particularly important in analyzing and optimizing embedded software. The importance and complexity of carefully analyzing scheduling strategies are high even for the restricted SDF model, where scheduling decisions have a major impact on most key implementation metrics [21]. The incorporation of dynamic dataflow features makes the scheduling problem more critical since application behaviors are less predictable, and more difficult to understand through analytical methods.

EIDF is based on a formalism in which actors execute through dynamic transitions among modes, where each mode has "synchronous" (constant production/-consumption rate behavior), but different modes can have differing dataflow rates. Unlike other forms of mode-oriented dataflow specification, such as stream-based functions (see Section 5), SDF-integrated starcharts (see Chapter 36), SysteMoc (see Chapter 36), and CAL (see Section 6), EIDF imposes a strict separation between fireability checking (checking whether or not the next mode has sufficient data to execute), and mode execution (carrying out the execution of a given mode). This allows for lightweight fireability checking, since the checking is completely separate from mode execution. Furthermore, the approach improves the predictability of mode executions since there is no waiting for data (blocking reads) — the time required to access input data is not affected by scheduling decisions or global dataflow graph state.

For a given EIDF actor, the specification for each mode of the actor includes the number of tokens that is consumed on each input port throughout the mode, the number of tokens that is produced on each output port, and the computation (the *invoke function*) that is to be performed when the actor is invoked in the given mode. The specified computation must produce the designated number of tokens on each output port, and it must also produce a value for the *next mode* of the actor, which determines the number of input tokens required for and the computation to be performed during the next actor invocation. The next mode can in general depend on the current mode as well as the input data that is consumed as the mode executes.

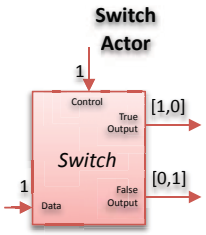
At any given time between mode executions (actor invocations), an enclosing scheduler can query the actor using the *enable function* of the actor. The enable function can only examine the number of tokens on each input port (without consuming any data), and based on these "token populations", the function returns a Boolean value indicating whether or not the next mode has enough data to execute to completion without waiting for data on any port.

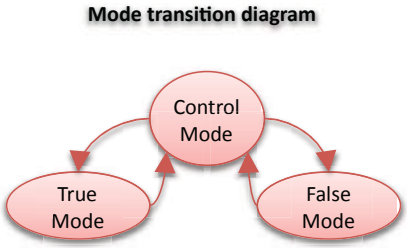
The set of possible next modes for a given actor at a given point in time can in general be empty or contain one or multiple elements. If the next mode set is empty (the next mode is `null`), then the actor cannot be invoked again before it is somehow reset or re-initialized from environment that controls the enclosing dataflow graph. A `null` next mode is therefore equivalent to a transition to a mode that

requires an infinite number of tokens on an input port. The provision for multi-element sets of next modes allows for natural representation of non-determinism in EIDF specifications.

When the set of next modes for a given actor mode is restricted to have at most one element, the resulting model of computation, called *core functional dataflow* (*CFDF*), is a deterministic, Turing complete model [9]. CFDF semantics underlie the *functional DIF* simulation environment for behavioral simulation of signal processing processing applications. Functional DIF integrates CFDF-based dataflow graph specification using the *dataflow interchange format* (*DIF*), a textual language for representing DSP-oriented dataflow graphs, and Java-based specification of intra-actor functionality, including specification of enable functions, invoke functions, and next mode computations [9].

Figure 15 and Figure 16 illustrate, respectively, the design of a CFDF actor and its implementation in functional DIF. This actor provides functionality that is equivalent to the Boolean dataflow switch actor described in Section 4.





Production & consumption behavior of modes

Mode	Control Input	Data Input	True Output	False Output
Control Mode	1	0	0	0
True Mode	0	1	1	0
False Mode	0	1	0	1

Fig. 15 An illustration of the design of a `switch` actor in CFDF.

Constructor

(modes and dataflow behavior defined)

```
public CFSwitch() {
       _control       = addMode("control");
       _control_true  = addMode("control_true");
       _control_false = addMode("control_false");

       _data_in    = addInput("data_in");
       _control_in = addInput("control_in");
       _true_out   = addOutput("true_out");
       _false_out  = addOutput("false_out");.

       _control.setConsumption(_control_in, 1);
       _control_true.setConsumption(_data_in, 1);
       _control_true.setProduction(_true_out, 1);
       _control_false.setConsumption(_data_in, 1);
       _control_false.setProduction(_false_out, 1);

}
```

Invoke Function

(performs action and determines next mode)

```
public CoreFunctionMode
              invoke(CoreFunctionMode mode){
       if (_init==mode) {
           return _control;
       }
       if(_control==mode) {
           if((Boolean)pullToken(_control_in)) {
              return _control_true;
           } else {
              return _control_false;
           }
       }
       if(_control_true==mode) {
           Object obj = pullToken(_data_in);

           pushToken(_true_out, obj);
           return _control;
       }
       if(_control_false==mode) {
           Object obj = pullToken(_data_in);

           pushToken(_false_out, obj);
           return _control;
       }
}
```

Enable Function

(determines if firing condition is met)

```
 public boolean enable(CoreFunctionMode mode){
       if (_control==mode) {
           if(peek(_control_in)>0) {
              return true;
           }
           return false;
       }else if (_control_true==mode){
           if(peek(_data_in)>0) {
              return true;
           }
           return false;
       } else if(_control_false==mode) {
         if(peek(_data_in)>0) {
              return true;
         }
         return false;
       }
       return false;
}
```

Fig. 16 An implementation of the `switch` actor design of Figure 15 in the functional DIF environment.

9 Summary

In this chapter, we have reviewed several DSP-oriented dataflow models of computation that are oriented towards representing multi-dimensional signal processing and dynamic dataflow behavior. As signal processing systems are developed and deployed for more complex applications, exploration of such generalized dataflow modeling techniques is of increasing importance. This chapter has complemented the discussion of Chapter 30, which focuses on the relatively mature class of decidable dataflow modeling techniques, and builds on the dynamic dataflow principles introduced in certain specific forms in Chapter 34 and Chapter 36.

References

1. S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet. Overview of the MPEG reconfigurable video coding framework. *Journal of Signal Processing Systems*, 2010. DOI:10.1007/s11265-009-0399-3.
2. H. Kee, I. Wong, Y. Rao, and S. S. Bhattacharyya. FPGA-based design and implementation of the 3GPP-LTE physical layer using parameterized synchronous dataflow techniques. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 1510–1513. Dallas, Texas, March 2010.
3. R. Gu, J. Janneck, M. Raulet, and S. S. Bhattacharyya. Exploiting statically schedulable regions in dataflow programs. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 565–568. Taipei, Taiwan, April 2009.
4. W. Plishker, N. Sane, and S. S. Bhattacharyya. A generalized scheduling approach for dynamic dataflow applications. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 111–116. Nice, France, April 2009.
5. W. Plishker, N. Sane, and S. S. Bhattacharyya. Mode grouping for more effective generalized scheduling of dynamic dataflow applications. In *Proceedings of the Design Automation Conference*, pages 923–926. San Francisco, July 2009.
6. H. Berg, C. Brunelli, and U. Lucking. Analyzing models of computation for software defined radio applications. In *Proceedings of the International Symposium on System-on-Chip*, 2008.
7. Y. Lin, Y. Choi, S. Mahlke, T. Mudge, and C. Chakrabarti. A parameterized dataflow language extension for embedded streaming systems. In *Proceedings of the International Symposium on Systems, Architectures, Modeling and Simulation*, pages 10–17, July 2008.
8. H. Nikolov and E. F. Deprettere. Parameterized stream-based functions dataflow model of computation. In *Proceedings of the Workshop on Optimizations for DSP and Embedded Systems*, 2008.
9. W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya. Functional DIF for rapid prototyping. In *Proceedings of the International Symposium on Rapid System Prototyping*, pages 17–23. Monterey, California, June 2008.
10. G. Roquier, M. Wipliez, M. Raulet, J. W. Janneck, I. D. Miller, and D. B. Parlour. Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, October 2008.
11. M. Ko, C. Zissulescu, S. Puthenpurayil, S. S. Bhattacharyya, B. Kienhuis, and E. Deprettere. Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation. *IEEE Transactions on Signal Processing*, 55(6):3126–3138, June 2007.
12. J. Keinert, C. Haubelt, and J. Teich. Modeling and analysis of windowed synchronous algorithms. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, May 2006.
13. S. Saha, S. Puthenpurayil, and S. S. Bhattacharyya. Dataflow transformations in high-level DSP system design. In *Proceedings of the International Symposium on System-on-Chip*, pages 131–136. Tampere, Finland, November 2006. Invited paper.
14. S. Neuendorffer and E. Lee. Hierarchical reconfiguration of dataflow models. In *Proceedings of the International Conference on Formal Methods and Models for Codesign*, June 2004.
15. J. Eker and J. W. Janneck. CAL language report, language version 1.0 — document edition 1. Technical Report UCB/ERL M03/48, Electronics Research Laboratory, University of California at Berkeley, December 2003.
16. B. Kienhuis and E. F. Deprettere. Modeling stream-based applications using the SBF model of computation. *Journal of Signal Processing Systems*, 34(3):291–299, 2003.
17. T. Stefanov and E. Deprettere. Deriving process networks from weakly dynamic applications in system-level design. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 90–96, October 2003.
18. P. K. Murthy and E. A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(8):2064–2079, August 2002.

19. E. D. Willink, J. Eker, and J. W. Janneck. Programming specifications in CAL. In *Proceedings of the OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2002.
20. B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, October 2001.
21. S. S. Bhattacharyya, R. Leupers, and P. Marwedel. Software synthesis and code generation for DSP. *IEEE Transactions on Circuits and Systems — II: Analog and Digital Signal Processing*, 47(9):849–875, September 2000.
22. S. Haykin. *Adaptive Filter Theory*. Prentice Hall, 1996.
23. S. S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee. Generating compact code from dataflow specifications of multirate signal processing algorithms. *IEEE Transactions on Circuits and Systems — I: Fundamental Theory and Applications*, 42(3):138–150, March 1995.
24. J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model*. Ph.D. thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, September 1993.
25. G. R. Gao, R. Govindarajan, and P. Panangaden. Well-behaved programs for DSP computation. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, March 1992.
26. J. Annevelink. *HIFI: A Design Method for Implementing Signal Processing Algorithms on VLSI Processor Arrays*. Ph.D. thesis, Delft University of Technology, Department of Electical Engineering, Delft, The Netherlands, 1988.
27. J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.