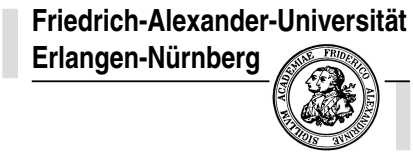


Windowed Synchronous Data Flow

Joachim Keinert, Christian Haubelt, Jürgen Teich

Department of Computer Science 12
Hardware-Software-Co-Design
University of Erlangen-Nuremberg
Am Weichselgarten 3
D-91058 Erlangen, Germany

Co-Design-Report 02 - 2005

January 17, 2006

Contents

1 Introduction

With the evolution in micro-electronics, digital systems can perform more and more complex functions. This, however, goes along with increased design complexity and as a consequence longer development periods and higher costs. Yet as markets require shorter and shorter innovation cycles, high level synthesis is often considered as one possibility to escape the complexity trap.

In order to obtain efficient high-level system synthesis results, various optimizations on different levels of abstractions have to be performed. This needs corresponding models expressing the necessary information in a as realistic manner as possible.

For throughput and buffer analysis as well as for operation ordering, data flow models are well suitable. Many different approaches comprising static and dynamic models have been presented in the past.

For image processing systems, local operators or window operators play a fundamental role. Many filtering operations as for instance edge detection, noise reduction, correlation or wavelet transform are part of this algorithm class. As a consequence, great effort should be performed in order to obtain precise models allowing for correct analysis and efficient synthesis of the desired applications.

Most window algorithms are of static nature and realized on multi-dimensional data. Such multi-dimensional applications can in principle be represented by one-dimensional models of computation such as *Synchronous Data Flow (SDF) [LM87]* or *Cyclo Static Data Flow (CSDF)* [BELP96]. However, as stated in [ML02], such an approach may be awkward, and much important knowledge about the algorithm as for instance the contained parallelism gets lost.

In order to cope with those problems, *Multidimensional Synchronous Data Flow* (MDSDF) was proposed in [ML02]. However, we will show, that the latter one does not allow for "overlapping windows" as it is required for many local operations applied in image processing algorithms.

Consequently, this report develops an extension to MDSDF, the so called *Windowed Synchronous Data Flow (WSDF)* model allowing for overlapping, multi-dimensional windows.

Section 2 introduces some important notations which are used throughout the document. Section 3 presents related work, especially already existing static data flow models. In Section 4, we investigate on how efficiently well-known image processing algorithms based on sliding windows can be modeled by currently known models of computation. Section 5 develops the new *Windowed Synchronous Data Flow*. Section 6 finally concludes this report with a summary and some perspectives.

2 Notation

$\mathbb{N}$	$\mathbb{N}$ is the set of natural numbers not containing zero.
$\mathbb{N}_0$	$\mathbb{N}_0$ is the set of natural numbers including zero.
$\langle \overrightarrow{a}, \overrightarrow{b} \rangle$	Dot product between the vectors $\overrightarrow{a}$ and $\overrightarrow{b}$
$scm\,(a, b)$	Smallest common multiple of a and b
$scm\,(X)$	Smallest common multiple of all elements $x \in X$. If $X = \emptyset$, then we define $scm\,(X) = 1$
$gcd\,(a, b)$	Greatest common divisor of a and b
$\lceil x \rceil$	Ceiling function: This indicates the smallest integer not exceeded by x.

3 Related Work

In the past, many different data flow models of computation have been proposed, as for instance *Synchronous Data Flow (SDF)* [LM87], *Cyclo Static Data Flow (CSDF)* [BELP96], *Fractional Rate Data Flow (FRDF)* [OH02], *Multidimensional Synchronous Data Flow (MDSDF)* [Che94], *Generalized Multidimensional Synchronous Data Flow (GMDSDF)* [ML02], *Parameterized Synchronous Data Flow (PSDF)* [Bha99], *Boolean Data Flow (BDF)* [Buc93] or *Cyclo Dynamic Data Flow (CDDF)* [WELP96]. Among other criteria, they can be classified by their capability to represent dynamic actor behavior. BDF or CDDF for instance are such models. PSDF introduces a system of parameter propagation and, as a consequence, allows for the description of some restricted amount of dynamic behavior.

As many of the considered image processing algorithms as edge detection filters or wavelet transform are of static nature, we will restrict our description to non-dynamic models.

3.1 Synchronous Data Flow

Synchronous Data Flow is a static, multi rate model of computation. Its graph consists of networks of actors connected by arcs that carry data.

Definition 3.1 *SDF-Graph*

An SDF graph is a tuple $G = (V, E, p, c, d)$. V is the set of vertices, also called actors, $E \subseteq V \times V$ the set of edges connecting the actors and transporting data elements in form of tokens. The source of an edge is denoted by $src(e)$, the sink by $snk(e)$. Each edge can have a positive delay $d(e)$ which corresponds to the number of initial tokens stored on the edge

$$d : E \to \mathbb{N}_0,\ e \mapsto d(e)$$

The number of tokens produced by $src(e)$ per invocation is specified by the function $p(e)$

$$p : E \to \mathbb{N},\ e \mapsto p(e)$$

The number of tokens consumed by $snk(e)$ per firing is defined by $c(e)$

$$c : E \to \mathbb{N},\ e \mapsto c(e)$$

Figure 1 shows an example graph with two actors. To each edge, an edge buffer is assigned storing arriving tokens. If an actor is executed or *fired*, $c(\alpha)$ tokens are removed from the buffer of each input edge α and $p(\beta)$ tokens are appended to the buffer of each output edge β.

An actor is *fireable*, if there are enough input tokens available in order to perform the actor functionality. The output buffers are considered to be of infinite size.

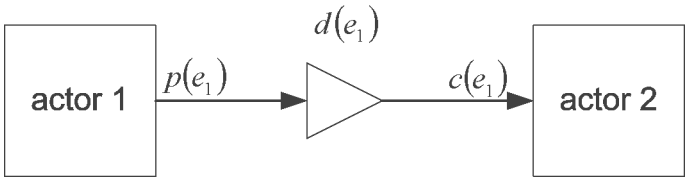


Figure 1: Example SDF Graph with two actors and one delay element

3.1.1 SDF Graph Schedule

Definition 3.2 *SDF-Schedule*

A schedule *for a SDF-Graph G is a sequence of actor firings $S = [v_1, v_2, \ldots]$. The schedule is repeated periodically and can be of finite or infinite length. Each element of S is an invocation of the corresponding actor $v_i \in V$. Each repetition of the schedule is called a* schedule period.

Definition 3.3 *SDF-Graph State*

As an SDF graph is executed in invocations of a schedule, the state of a graph *can be represented by the number of tokens stored on the edge buffers of the graph.*

Definition 3.4 *Periodic Schedules*

A periodic schedule *is a finite schedule which invokes each actor of the graph G at least one time and which does not cause a net graph state change.*

Definition 3.5 *Valid Schedule*

A valid schedule *is a periodic schedule which does not cause a graph deadlock. A deadlock occurs, if an actor planned for execution by schedule S is not fireable.*

Definition 3.6 *Consistency of SDF graph*

A SDF graph is called consistent, if and only if it has at least one valid schedule.

3.1.2 Balance Equation

As stated in Definition 3.4, a periodic schedule S does not cause any net graph state change. Consequently, the number of produced and consumed tokens on an edge must be identical.

The number of times an actor is fired in a valid schedule is represented by an integer vector $\overrightarrow{r_G}$, indexed by the number of actors in G. $\overrightarrow{r_G}(v)$ defines the number of times, actor v is fired.

A schedule S does not change the state of a graph G, if and only if

$$\forall e \in E : \overrightarrow{r}(src(e)) \cdot p(e) = \overrightarrow{r}(snk(e)) \cdot c(e) \tag{1}$$

The equations for the different edges of the graph can be combined to a matrix equation

$$\Gamma_G \cdot \overrightarrow{r_G} = \overrightarrow{0} \tag{2}$$

Definition 3.7 *Topology Matrix*

$$\Gamma_G = \begin{bmatrix} \Gamma_{1,1} & \cdots & \Gamma_{1,v} & \cdots & \Gamma_{1,|V|} \\ \vdots & & \vdots & & \vdots \\ \Gamma_{e,1} & \cdots & \Gamma_{e,v} & \cdots & \Gamma_{e,|V|} \\ \vdots & & \vdots & & \vdots \\ \Gamma_{|E|,1} & \cdots & \Gamma_{|E|,v} & \cdots & \Gamma_{|E|,|V|} \end{bmatrix}$$

is called topology matrix *and describes the token consumption and production of an actor v on edge e:*

$$\Gamma_{e,a} = \begin{cases} p(e) & \text{if } src(e) = v \text{ and } snk(e) \neq v \\ -c(e) & \text{if } snk(e) = v \text{ and } src(e) \neq v \\ p(e) - c(e) = 0 & \text{if } snk(e) = src(e) = v \\ 0 & \text{otherwise} \end{cases}$$

Definition 3.8 *Basic Repetition Vector*

The minimal, strictly positive integer vector $\overrightarrow{r_G}$ which solves equation 2, is called basic repetition vector. *It specifies, how often an actor has to be at least fired in a valid schedule. If understood from the context, the index G can be suppressed.*

Definition 3.9 *Minimal Periodic Schedule*

A minimal periodic schedule *is a periodic schedule invoking each actor $v \in V$ exactly $\overrightarrow{r_G}(v)$ times, whereas $\overrightarrow{r_G}$ is the basic repetition vector.*

Theorem 3.1 *For a connected SDF graph holds:*

$$|V| \geq rank(\Gamma_G) \geq |V| - 1$$

Proof 3.1 *See [LM87].*

Theorem 3.2 *For a connected SDF graph holds:*

$$\overrightarrow{r_G} \text{ exists } \Leftrightarrow rank(\Gamma_G) = |V| - 1$$

Proof 3.2 *See [LM87].*

3.1.3 SDF Deadlock Analysis

Even when the balance equation of an SDF graph has a strictly positive integer solution, it might not possible to find a valid schedule, because the graph deadlocks. In order to detect such situations, Lee proposes to construct a single-processor schedule for one iteration of the multirate graph ([LM87]).

Starting with an empty schedule $S = []$, we select a fireable actor and add it to the schedule. This is repeated, until each actor $v \in V$ occurs exactly $\overrightarrow{r_G}(v)$ times in the schedule S. If during this process a situation occurs, where no actor is fireable anymore, then the graph deadlocks whatever schedule will be applied, because the graph behavior is independent of the

actor invocation order. The complexity of this approach is non-polynomial. In oder words, $\mathcal{O} > \mathcal{O}\left((|V| + |E|)^n\right)$.

Karp and Miller ([KM66]) propose an alternative algorithm checking the termination of loops. If all loops in a consistent graph are nonterminating, then a valid periodic schedule can be found. For applications, where the number of existing loops in a graph is limited ([BELP96]), this might be an interesting alternative to the construction of a complete single processor schedule. Especially if a graph should be modified, only concerned loops have to be evaluated again, whereas the construction of a single processor schedule must be repeated completely.

3.2 Multidimensional Data Flow (MDSDF)

The SDF model presented in Section 3.1 only allows for one-dimensional tokens. Applications processing multidimensional data can often be advantageously described by a multidimensional model of computation ([ML02]).

Definition 3.10 *MDSDF-Graph*

A MDSDF graph is a tuple $G = (V, E, p, c, d)$. V is the set of vertices, also called actors, $E \subseteq V \times V$ the set of edges connecting the actors and transporting data elements in form of tokens with dimension n. Each edge can have a positive delay $d(e)$ with dimension n:

$$d : E \to \mathbb{N}_0^n,\ e \mapsto \overrightarrow{d}(e)$$

The number of data elements produced by $src(e)$ per invocation is specified by the function $p(e)$

$$p : E \to \mathbb{N}^n,\ e \mapsto \overrightarrow{p}(e)$$

*with $\langle \overrightarrow{p}(e), \overrightarrow{e_i} \rangle$ corresponding to the number of data elements produced in dimension i. $\overrightarrow{e_i}$ are the Cartesian base vectors and must not be confused with the edge index e. Supposing $n = 2$ dimensions, the first component of a vector specifies the number of **columns**, the second component the number of **lines**, if not mentioned otherwise.*

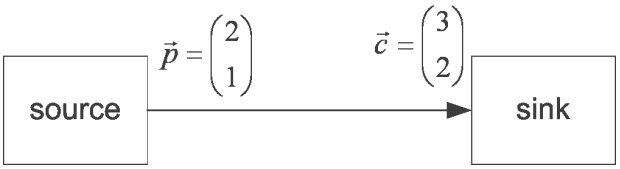
The number of data elements consumed by $snk(e)$ per firing is defined by $\overrightarrow{c}(e)$

$$c : E \to \mathbb{N}^n,\ e \mapsto \overrightarrow{c}(e)$$

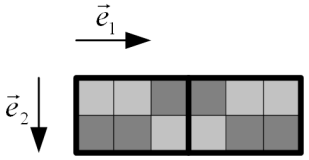
For the MDSDF model defined in [ML02], an actor is fireable, if and only if it is fireable in each dimension $1 \leq i \leq n$. The token consumption and production for a certain dimension i is independent of the rest of the dimensions.[1]

If we change the number of consumed or produced tokens in dimension i, then the number of actor invocations for the other dimensions stays unchanged. As a consequence, we are getting one balance equation per dimension.

[1] Generalized Multidimensional Synchronous Data Flow allows for more complex firing relations and is not considered in this report.



(a) Example MDSDF Graph



(b) Resulting token flow

Figure 2: MDSDF grouping of tokens. $\overrightarrow{e_1}$and $\overrightarrow{e_2}$ show the orientation of the supposed base vectors. This corresponds to an actor firing from left to right and from top to bottom.

$$\Gamma_i \cdot \overrightarrow{r_i} = \overrightarrow{0} \quad \forall i = 1, \ldots, n \tag{3}$$

with

$$\Gamma_{e,a,i} = \begin{cases} \langle \overrightarrow{p}\,(e), \overrightarrow{e_i} \rangle & \text{if } src\,(e) = a \text{ and } snk\,(e) \neq a \\ -\langle \overrightarrow{c}\,(e), \overrightarrow{e_i} \rangle & \text{if } snk\,(e) = a \text{ and } src\,(e) \neq a \\ \langle \overrightarrow{p}\,(e) - \overrightarrow{c}\,(e), \overrightarrow{e_i} \rangle = 0 & \text{if } snk\,(e) = src\,(e) = a \\ 0 & \text{otherwise} \end{cases}$$

3.2.1 Grouping of Tokens

Figure 2 shows an example MDSDF graph. As it can be seen, the produced tokens do not have the same size than the consumed ones. Still worse, even a multiple of produced tokens does not correspond to one single consumed token.

As a consequence, the source actor has to fire multiple times to produce one sink token. In vertical direction, two firings of the source actor are necessary to "fill" a sink token. In horizontal direction, one and a half source actor firings are necessary. As half firings are not supported by the MDSDF model, two sink tokens are combined to form one unit.

The size of this unit is not explicitly specified in the model, but can be calculated from the given sizes for the produced and consumed tokens. Furthermore the firing order is neither specified. So, the application can decide to fire first vertically, then horizontally or the inverse. Even zigzag firing might be possible.

An important observation which we can make here is, that the actor firing in dimension $i = 1$ and $i = 2$ are completely independent. Even if we modified $\langle \overrightarrow{p}, \overrightarrow{e_2} \rangle$ and $\langle \overrightarrow{c}, \overrightarrow{e_2} \rangle$, the firing in horizontal direction would not change.

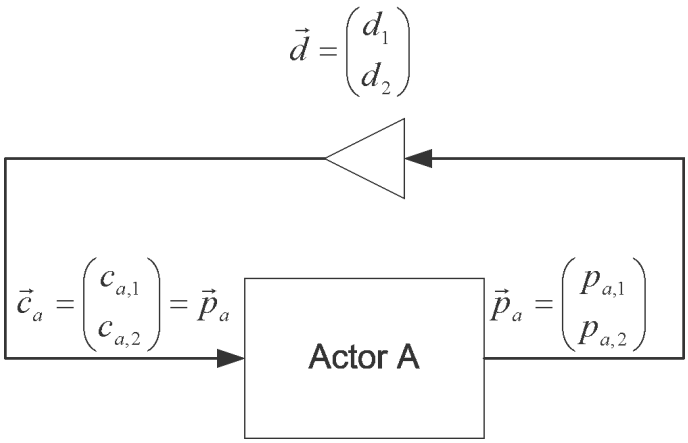


Figure 3: Self Loop in MDSDF

3.2.2 MDSDF Self-Loops

Delays in MDSDF are a complex issue. Different interpretations are possible ([Che94]), from which some however are difficult to encounter in practice. In the following, we assume the interpretation as given in [ML02], because the latter one permits self-loops which probably is the most frequent application.

Interpretation for $n = 2$ Dimensions Figure 3 shows an example MDSDF graph containing a self-loop. Vector $\overrightarrow{d}$ represents the delay element. Supposing $n = 2$ dimensions, it specifies the number of initial rows and columns available on the corresponding arc. This is shown in Figure 4.

Interpretation for more than two dimensions Having tokens with more than two dimensions, the interpretation of a MDSDF delay element is straight forward. $\langle \overrightarrow{d}, \overrightarrow{e_i} \rangle$ corresponds to the number of initial hyper-planes which are orthogonal to $\overrightarrow{e_i}$.

Sufficient and Necessary Condition for a Dead-Lock Free Self-Loop In [ML02], the authors prove the following relation for MDSDF:

Theorem 3.3 *Actor A deadlocks* $\Leftrightarrow \forall 1 \leq i \leq n$*:* $c_{a,i} > d_i$*, whereas* $\overrightarrow{p_a} = \overrightarrow{c_a}$.

3.3 Cyclo Static Data Flow

In the SDF model, the token consumption and production of an actor is constant for each invocation. The Cyclo Static Model enhances the data flow concept by allowing periodically changing token consumption and production patterns. An actor is executed in a sequence of phases each producing and consuming a constant number of tokens.

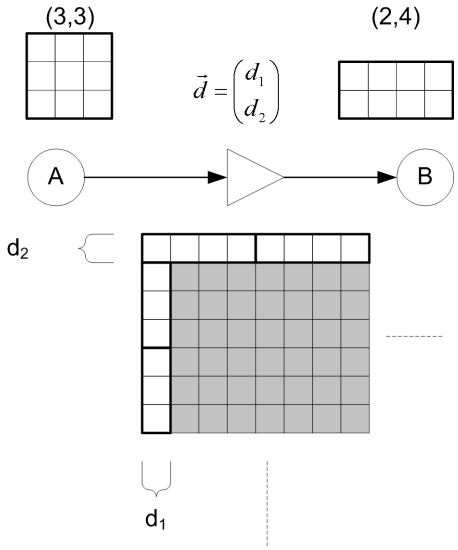


Figure 4: Interpretation of a MDSDF delay element with $n = 2$ dimensions

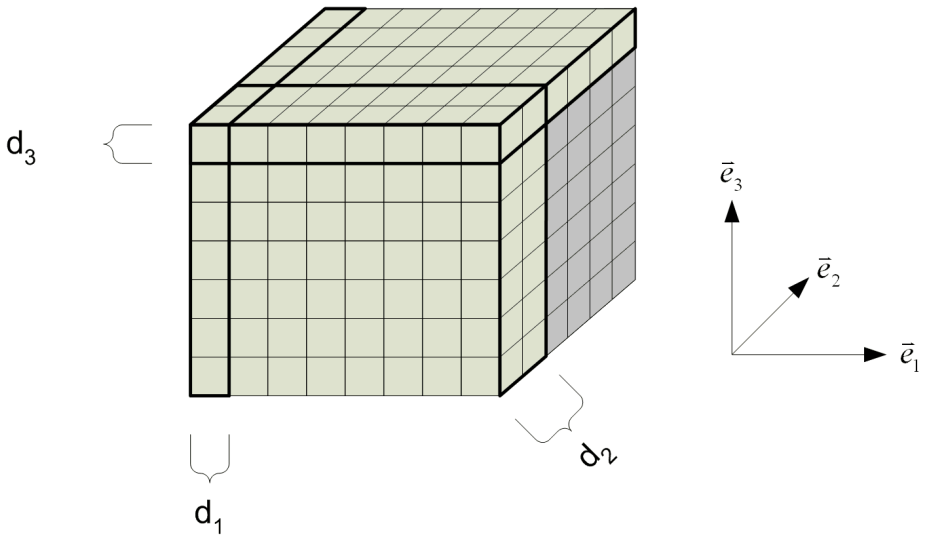


Figure 5: Interpretation of a MDSDF delay element with $n = 3$ dimensions

Definition 3.11 *CSDF-Graph*

A CSDF graph is a tuple $G = (V, E, L, p, c, d)$. V is the set of vertices, also called actors, $E \subseteq V \times V$ the set of edges connecting the actors and transporting data elements in form of tokens. Each edge can have a positive delay $d(e)$ which corresponds to the number of initial tokens stored on the edge

$$d : E \to \mathbb{N}_0,\ e \mapsto d(e)$$

The minimum period in the token consumption or production behavior of an actor a is given by the function

$$L : V \to \mathbb{N},\ v \mapsto L(v)$$

The number of tokens produced by $src(e)$ depends from the current phase number and is given by $p(e, s)$

$$p : E \times \{s \in \mathbb{N} : 0 \leq s < L(src(e))\} \to \mathbb{N},\ (e, s) \mapsto p(e, s)$$

The number of tokens consumed by $snk(e)$ at phase s is defined by $c(e, s)$

$$c : E \times \{s \in \mathbb{N} : 0 \leq s < L(snk(e))\} \to \mathbb{N},\ (e, s) \mapsto c(e, s)$$

During the k-th execution of $src(e)$, the token production is given by $p(e, (k-1)\ mod\, L(src(e)))$. The token consumption of $snk(e)$ can be calculated by $c(e, (k-1)\ mod\, L(snk(e)))$.

Definition 3.12 *CSDF-Graph State*

The state *of a CSDF graph can be represented by the number of tokens stored on the edge buffers of the graph and by the phase index of each graph actor.*

3.3.1 Balance Equation

As for SDF, a periodic schedule does not cause any net change in the graph state. As a consequence, for each edge the number of consumed and produced tokens must be identical. Furthermore, each actor must return into the same phase from which it started. The number of necessary actor invocations for such a periodic schedule is given by the repetition vector $\overrightarrow{r_G}$.

Theorem 3.4 *Balance Equation*

In a CSDF graph G, a repetition vector $\overrightarrow{r_G}$ is given by

$$\overrightarrow{r_G} = M \cdot \overrightarrow{q_G},\ \textit{with}\ M = \begin{pmatrix} L(v_1) & 0 & \cdots & 0 \\ 0 & L(v_2) & & \\ \vdots & & & \\ 0 & & & L(v_{|V|}) \end{pmatrix} \tag{4}$$

The number of invocations of actor v_i is given by $\overrightarrow{r_G}(v_i) = \langle \overrightarrow{r_G}, \overrightarrow{e_i} \rangle$.

Vector $\overrightarrow{q_G}$ is a strictly positive integer solution of the balance equation

$$\underbrace{\begin{bmatrix} \Gamma_{1,1} & \cdots & \Gamma_{1,v} & \cdots & \Gamma_{1,|V|} \\ \vdots & & \vdots & & \vdots \\ \Gamma_{e,1} & \cdots & \Gamma_{e,v} & \cdots & \Gamma_{e,|V|} \\ \vdots & & \vdots & & \vdots \\ \Gamma_{|E|,1} & \cdots & \Gamma_{|E|,v} & \cdots & \Gamma_{|E|,|V|} \end{bmatrix}}_{\Gamma} \cdot \overrightarrow{q} = \overrightarrow{0}$$

with

$$\Gamma_{e,v} = \begin{cases} \sum_{i=1}^{L(v)} p(e,i) & \text{if } v = src(e) \\ -\sum_{i=1}^{L(v)} c(e,i) & \text{if } v = snk(e) \\ \sum_{i=1}^{L(v)} p(e,i) - \sum_{i=1}^{L(v)} c(e,i) = 0 & \text{if } v = snk(e) = src(e) \\ 0 & \text{otherwise} \end{cases}$$

$\overrightarrow{q_G}(v_i) = \langle \overrightarrow{q_G}, \overrightarrow{e_i} \rangle$ is the number of times, a complete sequence of actor v_i is executed.

Proof 3.3 *See [BELP96].*

Equation (4) assures, that each actor returns to its initial phase.

Theorem 3.5 *For a connected CSDF graph holds:*

$$\overrightarrow{r} \text{ exists } \Leftrightarrow rank(\Gamma) = |V| - 1$$

Proof 3.4 *See [BELP96].*

3.4 Fractional Rate Data Flow

Implementations based on SDF graphs tend to have high memory requirements and large latencies due to non-optimal buffer management, especially if tokens correspond to data structures as for instance pixel arrays ([OH02]).

Fractional Rate Data Flow is a model of computation which improves this situation by allowing consumption of *fractional tokens*, that means only parts of a composite data structure.

Definition 3.13 *FRDF-Graph*

An FRDF graph is a tuple $G = (V, E, p, e, d)$. V is the set of vertices, also called actors, $E \subseteq V \times V$ the set of edges connecting the actors and transporting data elements in form of tokens. Each edge can have a positive delay $d(e)$ which corresponds to the number of initial tokens stored on the edge

$$d : E \to \mathbb{N}_0,\ e \mapsto d(e)$$

The number of tokens produced by $src(e)$ per invocation is specified by the function $p(e)$

$$p : E \to \mathbb{N}^2,\ e \mapsto p(e) = \begin{pmatrix} p_{num}(e) \\ p_{denom}(e) \end{pmatrix}$$

$p_{num}(e)$ is the number of (complete composite) tokens produced. This production however can be performed in a fractional manner. $p_{denom}(e)$ defines the fractional part of the composite token which is consumed for each actor invocation. For $p(e)$, the following short notation is introduced:

$$p(e) = \frac{p_{num}(e)}{p_{denom}(e)}$$

*This reminds to a fraction which however **must not be canceled**. Otherwise the meaning of the model would change.*

The number of tokens consumed by $snk(e)$ per firing is defined by $c(e)$ in an analog manner.

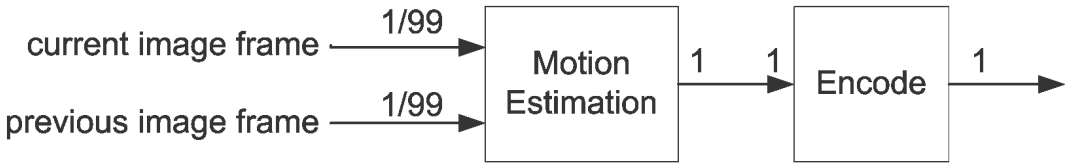
$$c : E \to \mathbb{N}^2,\ e \mapsto c(e) = \begin{pmatrix} c_{num}(e) \\ c_{denom}(e) \end{pmatrix} =: \frac{c_{num}(e)}{c_{denom}(e)}$$

Example 3.1 *Figure 6 shows an example taken from [OH02]. The actor* motion estimation *compares the content of the current and the previous image in a block based manner by tiling the image into blocks having only 1/99 of the input image size. Each of those blocks is encoded independently by the actor* encode*. The fractional consumption and production of the corresponding compound tokens lead to a very efficient buffer management ([OH02]).*

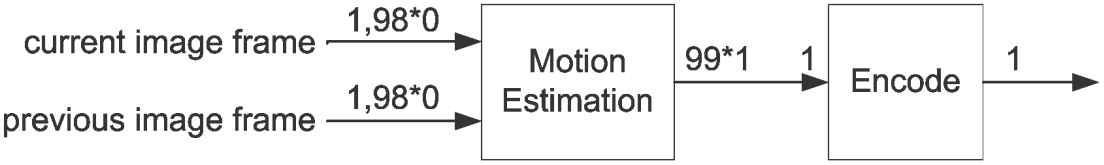
3.4.1 Token Orientation

FRDF only operates with abstract fractions, but does not tell how these fractions are constructed. As a consequence, efficient analysis and implementation needs further information about *token compatibility.*

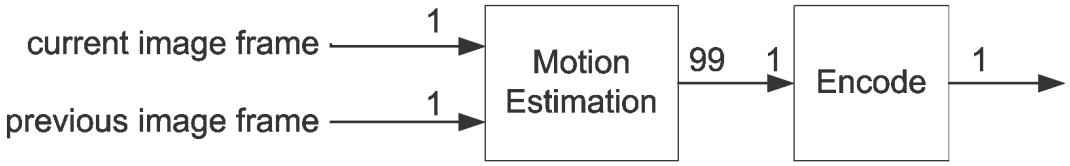
Figure 7 illustrates this problem: Although the source actor produces the composite tokens in fractions of a half token, and the sink consumes them in the same manner, the shown application cannot profit from the benefits of fractional token consumption, because the fraction interpretation does not coincide. As a consequence, the composite token has to be considered as atomic.



(a) FRDF model



(b) buffer memory equivalent CSDF model



(c) functional equivalent SDF graph

Figure 6: Block-Encoding of data generated by a motion estimation algorithm. In order to reduce the entropy which must be encoded, the previous frame is subtracted from the current one taking into account the contained motion. The result is cut into blocks covering $\frac{1}{99}$ of the original image size. These blocks are coded independently by the *Encode* block. In (a), the corresponding FRDF graph (a) is shown, (b) presents the CSDF graph (b) having the same memory requirements. It is supposed, that both the current and the previous image frame arrive "at once" (not in fractions). In both cases, the resulting blocks can be immediately consumed by the *encode* actor instead of waiting for the resting results of the complete input image. This leads to a very efficient buffer management on the edge connecting both actors. (c) shows the functional equivalent SDF model. Here the 99 blocks are produced during one actor invocation. As a consequence, all of them have to be stored before the Encode-Actor can start processing them.[2]

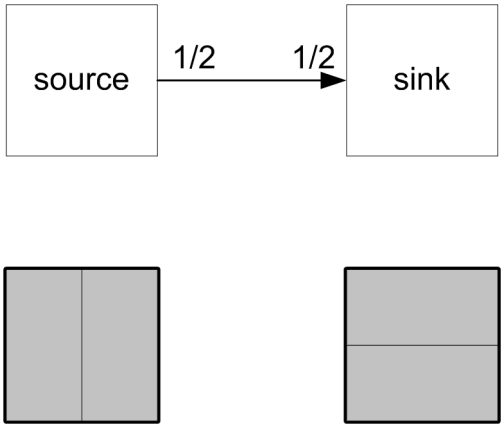


Figure 7: FRDF model with incompatible fractional token orientation. Both the source and the sink actor operate on half tokens. However, their interpretation of the fraction $\frac{1}{2}$ differs. The source actor produces the output token by complete columns, whereas the sink actor needs complete lines. In order to assure a correct behavior of the model, the composite token has to be considered as atomic.

4 Modeling of Image Processing Algorithms with Existing Models of Computation

Image processing systems contain algorithms of various types. One possible criterion for classification is their spatial locality. In this context, we distinguish global algorithms, point algorithms and local algorithms. The latter ones can often be implemented by sliding a window of a certain size over the image.

Such window operators are a fundamental part of every image processing system. This section investigates, how such operators can be modeled and analyzed by different existing models of computation. The considerations base on two different examples:

- A block builder functionality as it is used in the JPEG2000 standard
- An ordinary window operation as it occurs for instance during convolution

After a short introduction into these two image processing algorithms, Section 4.3 shows their representation in SDF and investigates the possibilities for analysis. Section 4.4 concentrates on CSDF, and Section 4.5 finally investigates on the MDSDF model of computation. Section 4.6 concludes with some remarks concerning FRDF.

4.1 The JPEG2000 Block Builder

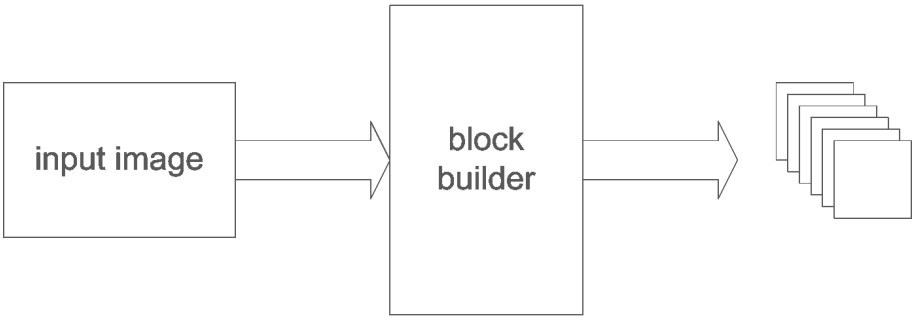
JPEG2000 ([ISO]) is an algorithm for compression of still images. The image pixels are decorrelated by means of a wavelet transform and coded by a sophisticated entropy coder. In order to increase the parallelism of the JPEG2000 algorithm and to facilitate the random access to image regions in the code stream, the wavelet coefficients are divided into independent blocks before performing the entropy coding.

The functionality of such a block-builder is shown in Figure 8. The input images representing the different subbands generated by the wavelet transform are tiled into equal sized blocks with height h_b and width w_b. Only at the borders, the resulting blocks might be smaller, if the input image width w respectively height h is not a multiple of the block width w_b respectively height h_b. Figure 8 supposes, that the upper left corner of the block grid coincides with the image borders in order to simplify the following data flow representations.

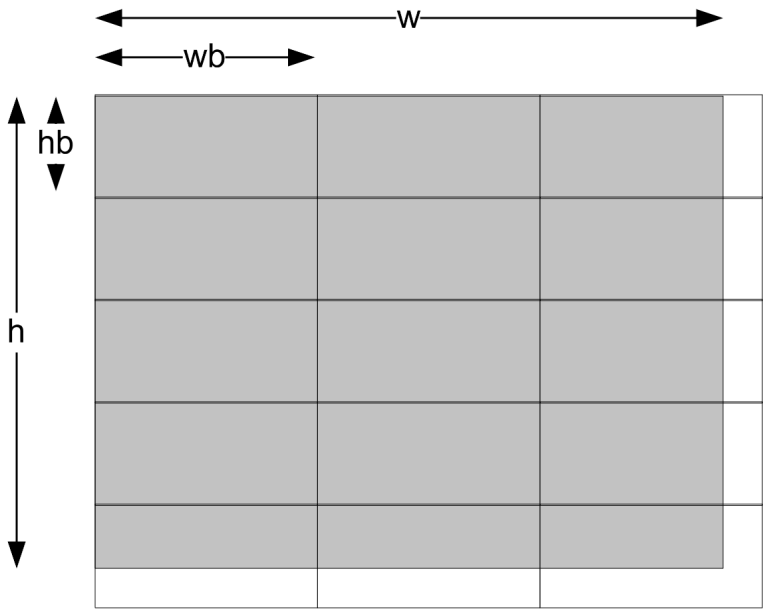
4.2 Convolution Windows

Convolution filters are widely applied in image processing applications for different purposes, as for instance edge detection, smoothing, wavelet transform or simple morphologic operations. The major difference between the JPEG2000 block builder and these kinds of algorithms is the overlapping of consecutive windows for the latter ones.

Figure 9 shows the geometry of a symmetric window kernel as it is assumed in the following. The hatched center pixel represents the reference pixel.



(a) principle of the JPEG2000 block builder



(b) Geometric representation of the image subdivision into blocks

Figure 8: Functionality of the JPEG2000 block builder. In part (b), the grey rectangle represents the set of wavelet coefficients of a subband, the grid shows the division into blocks. h and w represent the height respectively width of the image. The width of the resulting blocks is given by w_b, the height by h_b.

Mathematically, the convolution can be described by the following formula:

$$I_2(u,v) = \sum_{x=x_{w0}}^{x_{w1}} \sum_{y=y_{w0}}^{y_{w1}} g(x,y) \cdot I_1(u-x, v-y) = \sum_{x=-x_{w1}}^{-x_{w0}} \sum_{y=-y_{w1}}^{-y_{w0}} g(-x,-y) \cdot I_1(u+x, v+y) \tag{5}$$

whereas $I_1(u,v)$ is the input image, $I_2(u,v)$ the output image and $g(x,y)$ the rotated *filter kernel.*

The filter kernel is moved over the image in such a way, that the center pixel scans each input

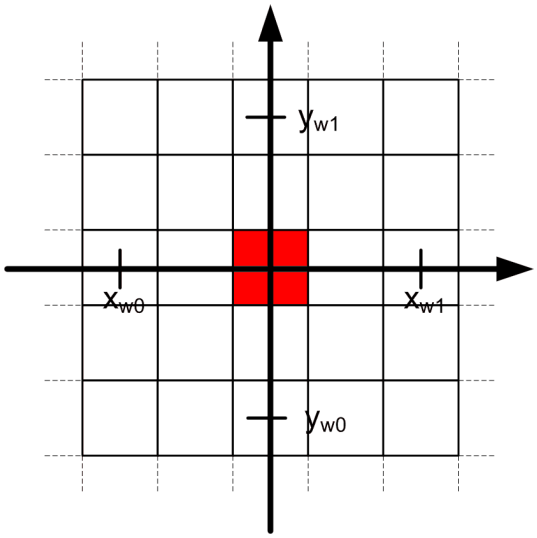


Figure 9: Symmetric convolution filter kernel. The red pixel represents the center of the window and has the same distance from the horizontal and vertical borders ($x_{w0} = -x_{w1}$, $y_{w0} = -y_{w1}$). The width of the window is $x_{w1} - x_{w0} + 1$, the height $y_{w1} - y_{w0} + 1$.

image pixel. As a consequence, for some window positions border processing is necessary in order to determine the values of the window pixels which are situated "outside" of the input image. To do so, several approaches exist:

- Symmetric extension of the input image
- Addition of a white or black border to the input image

Some applications avoid border processing by reducing the output image size.

Independent of the chosen algorithm for border processing, convolution shows cyclo static behavior. Before the first output pixel can be calculated, several lines respectively columns must be available. At the image end some output pixels do not need any further input pixels.

4.3 Modeling with Synchronous Data Flow (SDF)

In the following chapters, it is analyzed how the image processing algorithms presented above can be efficiently modeled by data flow representations. In more detail, the following points are considered:

- Derivation of necessary buffer space
- Extraction of contained parallelism
- Expressiveness and comprehensibility of the obtained models.

SDF is the most simple data flow model. It is fully static and can be analyzed during compile time for buffer memory requirements ([ALP97, BML99]) and deadlocks ([LM87, KM66]). As a consequence, it is possible to derive a near optimal schedule and buffer allocation. This however is only true, if the model represents efficiently the given problem.

Exactly the latter point is a major problem of SDF. As SDF only supports constant token production and consumption behavior, it is in general necessary to operate on whole images. This, however, leads to a waste of necessary memory. Especially for hardware implementations, this cannot be tolerated.

4.3.1 Convolution Filter

The convolution filter shows best the problems of SDF models for image processing algorithms and is illustrated in Figure 10.

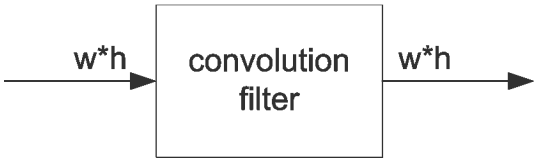


Figure 10: SDF model of a convolution filter. w is the image width, h the image height. Each output pixel depends from the local neighbor pixels belonging to the sliding window and can for instance be calculated by Equation (5).

As the convolution algorithm has cyclo-static behavior, we cannot create a model on pixel level. The actors have to consume and produce whole images. This, however leads to a much more pessimistic buffer memory analysis as it is required by the nature of the algorithm. For instance, applying a filter covering at maximum n involved image lines in a processing chain without loops and feedbacks, an efficient hardware implementation would be possible by storing only just n image lines, and not a complete image.

Furthermore, the SDF model does not expose any parallelism which might be contained in the algorithm. As we are not able to recognize how the output pixels depend from the input pixels, it is impossible to determine, if the output pixels can be calculated in parallel and which amount of data has to be duplicated for effective parallel processing.

4.3.2 JPEG2000 Block Builder

Figure 11 shows an SDF model for the JPEG2000 block builder.

In addition to the problems shown with the convolution algorithm, the block builder demonstrates another inefficiency: As we cannot model that some blocks are incomplete, this leads to inefficient data transmission behavior, because more pixels than effectively necessary are transported. As JPEG2000 supports more than one wavelet decomposition level, this overhead can get very remarkable.

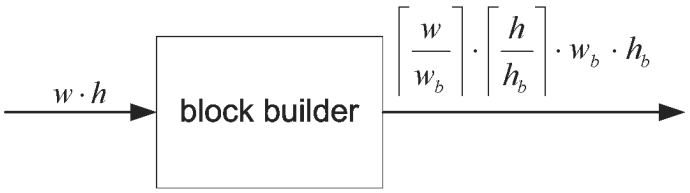


Figure 11: SDF model of the JPEG2000 block builder. w denotes the image width, h the image height. w_b and h_b define the block size.

4.4 Modeling with Cyclo Static Data Flow

One of the major inconveniences encountered in Section 4.3 has been the large buffer sizes due to the fact, that complete images have to be buffered.

Problems of this kind are known to be solved by Cyclo Static Data Flow. Besides reduced buffer requirements, such a description is much better suited for an implementation in hardware, as images are never transported as a single large block.

4.4.1 The JPEG2000 Block Builder

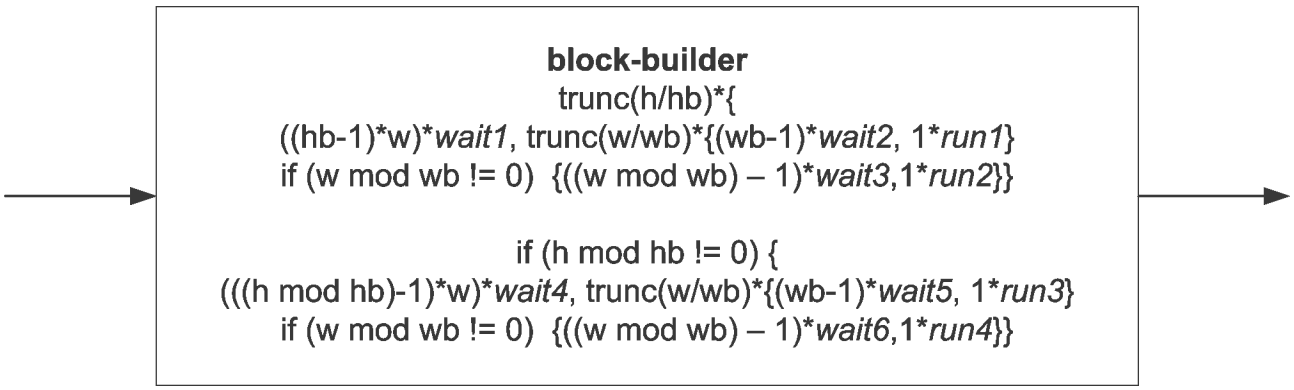


Figure 12: Extended CSDF model of the JPEG2000 block builder. w is the image width, h the image height. w_b and h_b define the block size. All parameters are constants which are supposed to be known at compile time.

Figure 12 shows a possible CSDF model of a block builder. The input image is read pixel by pixel. If all pixels for a block are available, the block is written out. Often, the resulting block is also written in pixels and not as a complete block. This, however, has not been taken into account in order to avoid a even more complex phase sequence.

The token consumption and production behavior is shown in Table 1.

In order to be able to represent the phase sequence, the CSDF representation in Figure 12 is extended by C-like syntactic constructs:

Phase	token consumption	token production
wait1	1	0
wait2	1	0
run1	1	$w_b \cdot h_b$
wait3	1	0
run2	1	$(w \, mod \, w_b) \cdot h_b$
wait4	1	0
wait5	1	0
run3	1	$(h \, mod \, h_b) \cdot w_b$
wait6	1	0
run4	1	$(w \, mod \, w_b) \cdot (h \, mod \, h_b)$

Table 1: Token consumption and production for the CSDF block builder model

- *trunc(h/hb)*{...}* executes the block contained between the braces $trunc\left(\frac{h}{h_b}\right)$ times, whereas $trunc$ corresponds to the well known C-function.
- *if (condition) {...}* only executes the block contained between the braces, if the condition evaluates to true.

As both the block and the image dimensions are supposed to be constant, all decisions and loops can be analyzed during compile time, so that the actor in Figure 12 is cyclo-static.

In comparison with SDF, the CSDF solution is much more economic concerning memory resources. In fact, we need only to store the input lines belonging to one single output block, if we suppose a processing chain without loops and feedbacks. In SDF, a token must correspond to a whole image due to border processing behavior. As a consequence, at least one input image must be stored in the edge buffer.

However, the CSDF-model shows the following problems:

- CSDF models are difficult to understand due to complex phase sequences
- From the model in Figure 12 it is not possible to derive the necessary buffer memory for storage of intermediate lines. This has to be modeled elsewhere, for instance by means of a corresponding refinement or by telling the compiler explicitly, that the above actor needs additional inside buffer memory.
- It does not show all degrees of parallelism. Due to the hidden buffer memory, we are not able to recognize, how the output pixels depend from the input pixels. As a consequence, we cannot decide, if it would be possible to implement this algorithm in a parallel fashion.
- CSDF models are not very general. In Figure 12 it is supposed, that the input image is generated line by line. Other production patterns require another model.

4.4.2 Convolution Filters

Figure 13 shows a possible CSDF model of a convolution filter. The input and output images are consumed and produced pixel by pixel. Before the first output pixel can be calculated, several input pixels must be already available (grey hatched zone in Figure 13(a). As the size of the input and the output image is identical, a termination phase is necessary, where output pixels are produced without consuming any input pixels.

The phase lengths can be calculated by the following formula:

$$\beta = \gamma = y_{w1} \cdot w + x_{w1},\ \alpha = w \cdot h - \beta$$

The meaning of these different variables is explained in Figure 13(a), where the hatched center pixel has the coordinates $(0, 0)$ and $y_{w1} = -y_{w0},\ x_{w1} = -x_{w0}$ due to supposed symmetry.

Besides the disadvantages shown for the JPEG2000 block builder, the CSDF model for the convolution given in Figure 13 has restricted generality. Especially in hardware, where a stream of images has to be processed, the termination and the preparation phase of consecutive images can be overlaid. This leads to the model shown in Figure 14.

As a consequence, we need different models for the same image processing algorithm. Especially for design space exploration, where we automatically want to determine the best solution in dependence of delay and throughput constraints, this is not well seen.

4.5 Modeling with Multidimensional Synchronous Data Flow

One of the major problems of the CSDF models presented in Chapter 4.4 is the missing possibility of exact buffer analysis due to hidden memory and due to the representation of a 2D problem by a 1D description. This is significantly improved by MDSDF.

4.5.1 The JPEG2000 Block Builder

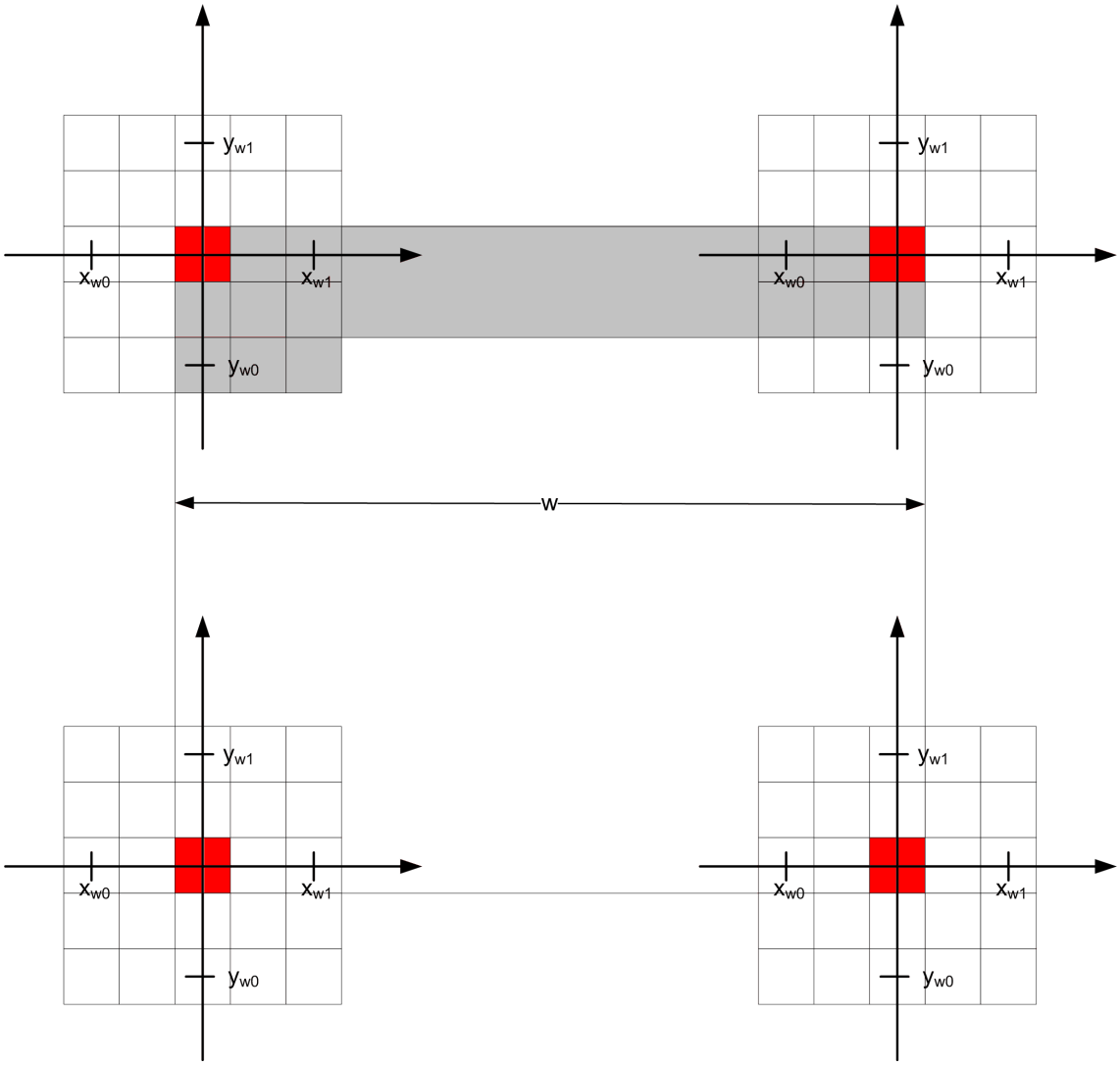
Supposing that the block borders coincide with the image borders, then the block builder actor can be modeled as shown in Figure 15.

The token consumption and production is illustrated in Figure 16 for a block size of 4x4.

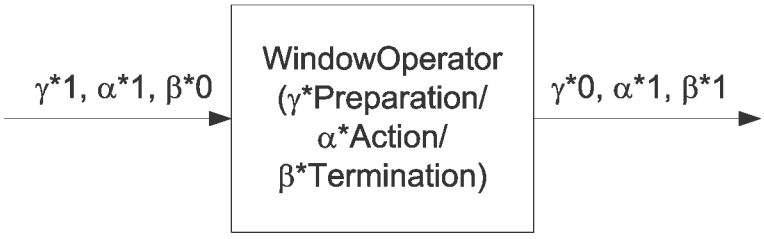
Although being a very elegant description, the interpretation of MDSDF does not always correspond to the desired application. If the image width, respectively the image height is not a multiple of the block width, respectively height, the modeled behavior is not that described in Figure 8. This can be seen in Figure 17.

4.5.2 Convolution Filter

As MDSDF only allows to model non-overlapping windows, the convolution filter cannot be represented by this model of computation. In [ML02], an extension to MDSDF is proposed in order to support arbitrary lattices for upsampling and downsampling operations. However the issue of overlapping windows is not considered. To the best of our knowledge, there does not exist any extension to MDSDF which allows for overlapping windows.



(a) Geometric illustration of the preparation and termination phase



(b) CSDF actor description

Figure 13: CSDF model of a convolution filter. $\beta = \gamma = y_{w1} \cdot w + x_{w1}$, $\alpha = w \cdot h - \beta$ The number of produced and consumed pixels can be calculated by $\gamma + \alpha = \alpha + \beta = w \cdot h$

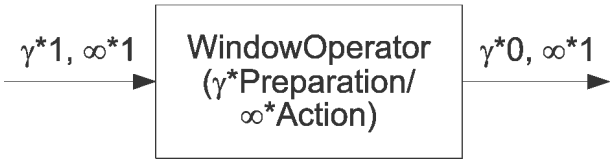


Figure 14: CSDF model of a convolution filter without termination phase

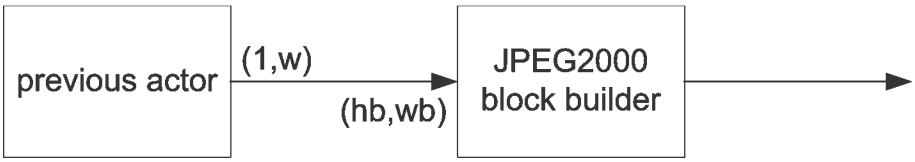


Figure 15: MDSDF model of the JPEG2000 block builder. w is the image width, hb and wb define the block extensions.

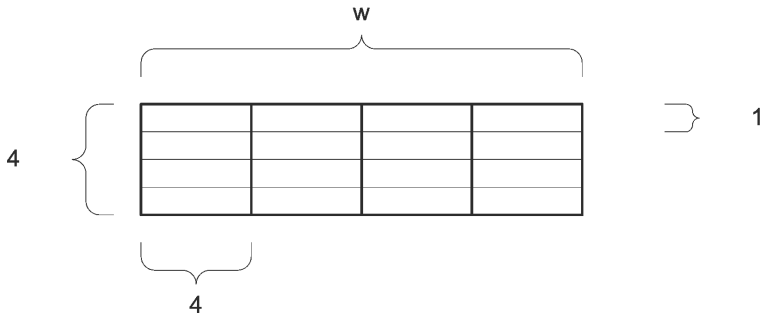


Figure 16: MDSDF token flow for a block size of 4x4

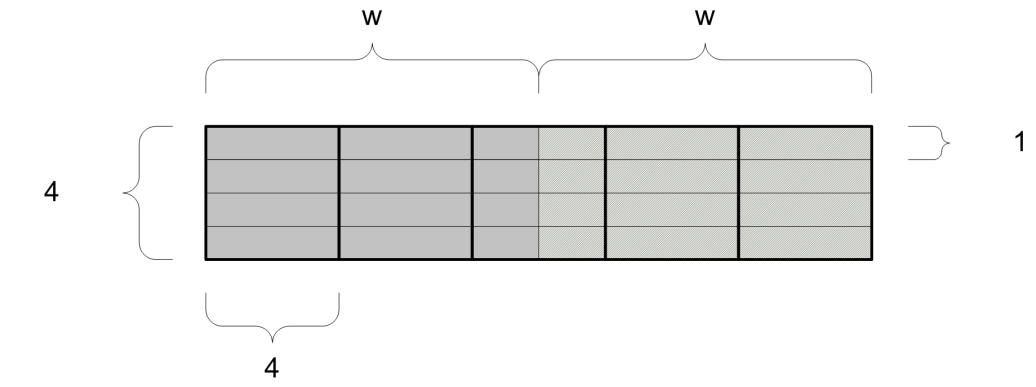
4.6 Modeling with Fractional Rate Dataflow (FRDF)

Due to the possibility of fractional token consumption and production, FRDF leads in general to very efficient implementations. However, similar to MDSDF, we do not know any possibility to model overlapping windows. As a consequence, the description of a convolution filter is difficult, especially if it should include the cyclo-static behavior as described in Chapter 4.4.2.

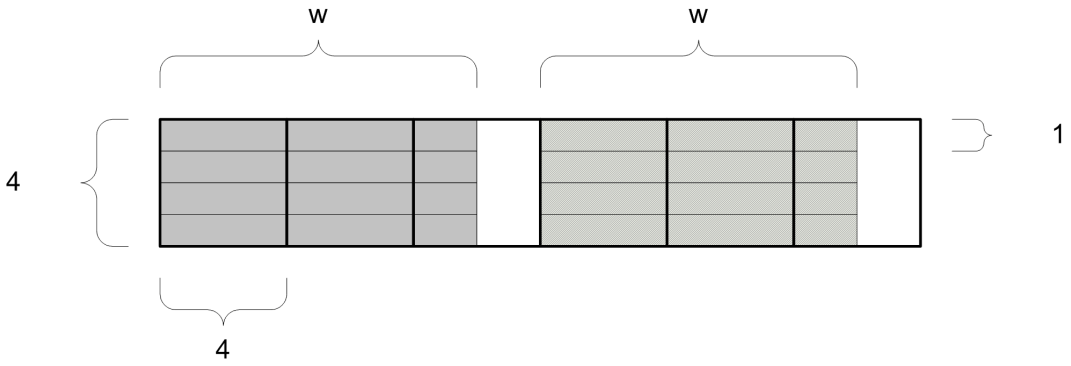
4.7 Summary

Window operators are fundamental components of image processing systems. However, as the analysis showed, to our knowledge no model of computation exists which can describe the behavior of such operators in a simple and precise manner. In dependency of the chosen model of computation, the following disadvantages have been found:

- Missing information about real buffer needs



(a) modeled behaviour



(b) desired behaviour

Figure 17: MDSDF token flow for non-coinciding image and block borders

- Complex phase sequences
- Incomplete exposure of contained parallelism
- Insufficient expressiveness
- Description not general enough
- Not very intuitive model

However, as window operations are part of almost every image processing system, it seems necessary to develop a more suited formalism. This by the way is what has been done in the SA-C Language [HB01] on a textural basis. As data flow representation is a good base for system specification, analysis, optimization and implementation, the following sections try to increase the expressiveness of MDSDF in order to efficiently model image processing window operators.

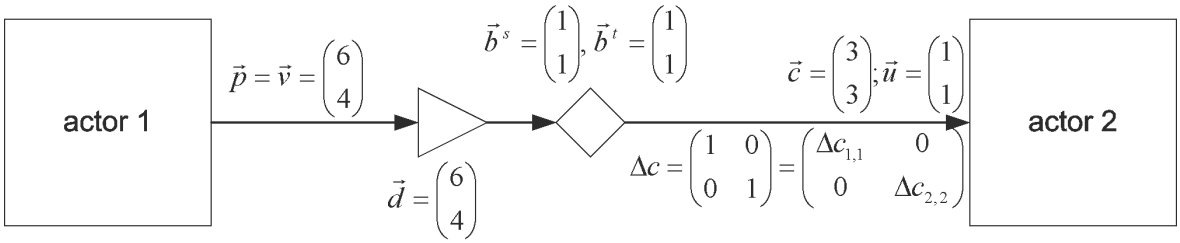


Figure 18: Exemplary WSDF edge showing the notation in a WSDF graph

5 Windowed Synchronous Data Flow

Local operators are an important algorithm class for many image processing applications. They often are of static nature and operate on multidimensional data. Efficient analysis and synthesis of such algorithms need a model of computation representing their major properties as parallelism, buffer requirements and invocation counts.

MDSDF would be a good solution, however it does not allow for overlapping sliding windows. *Windowed Synchronous Data Flow (WSDF)* is an extension of MDSDF in order to find a remedy to this situation.

5.1 WSDF-Graph

In comparison with the MDSDF model, the progression of overlapping windows needs an additional parameter which represents the so called *sampling grid*. In the style of [ML02], its description is performed by specifying a *sampling matrix*. As, however, most of the applications can be sufficiently modeled by a rectangular window progression, we restrain here to diagonal sampling matrices, in order to limit complexity.

Definition 5.1 *WSDF graph*
A WSDF graph *is a tuple* $G = \left(V, E, \overrightarrow{p}, \overrightarrow{v}, \overrightarrow{c}, \Delta c, \overrightarrow{u}, \overrightarrow{d}, \overrightarrow{b^s}, \overrightarrow{b^t}\right)$. V *is the set of* vertices*, also called* actors*,* $E \subseteq V \times V$ *the set of* edges *connecting the actors and transporting data elements in form of tokens with dimension* n*. The source of an edge is denoted by* $src\,(e)$*, the sink by* $snk\,(e)$*.*

The token production in a WSDF graph is specified by the functions $\overrightarrow{p}\,(e)$ *and* $\overrightarrow{v}\,(e)$ *(see Section 5.3), the token consumption is described by the functions* $\overrightarrow{c}\,(e)$*,* $\Delta c\,(e)$ *and* $\overrightarrow{u}\,(e)$ *(see Section 5.4). The function* $\overrightarrow{d}\,(e)$ *defines initial tokens on an edge and is described in Section 5.6.* $\overrightarrow{b^s}$*,* $\overrightarrow{b^t}$ *model border processing and are explained in Section 5.7.*

Figure 18 illustrates a single WSDF edge showing the notation used for a WSDF graph.

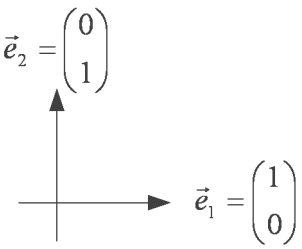


Figure 19: Example coordinate system with $n = 2$ dimensions used for interpretation

5.2 WSDF Tokens

In WSDF *tokens* are n-dimensional arrays of *data elements.* The model distinguishes between effective and virtual tokens. Their size is specified by an n-dimensional vector. For instance, having two-dimensional tokens, the first dimension of the vector specifies the number of columns, the second one the number of rows. The corresponding coordinate system is shown in Figure 19.

5.3 WSDF Token Production

Definition 5.2 *WSDF Token Production*
Each time, a source actor $src(e)$ is fired, it outputs an effective *token of constant size. The latter one is given by the function $\overrightarrow{p}(e)$*

$$\overrightarrow{p} : E \to \mathbb{N}^n,\ e \mapsto \overrightarrow{p}(e) =: \overrightarrow{p_e}$$

Effective tokens are combined to so called virtual tokens *of size $\overrightarrow{v}(e)$:*

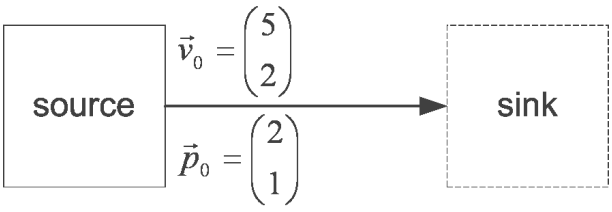
$$\overrightarrow{v} : E \to \mathbb{N}^n,\ e \mapsto \overrightarrow{v}(e) =: \overrightarrow{v_e}$$

Figure 20 shows an example for WSDF token production. Each time, the source actor is fired, it produces a token array with two columns and one line. These tokens are combined to a virtual token with five columns and two lines.

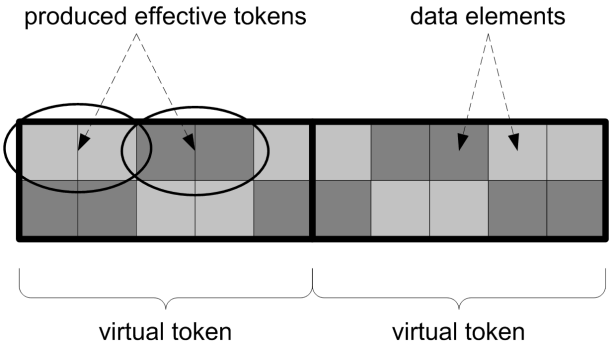
The idea behind virtual tokens is to form a unit of data elements belonging together. For instance, an image is composed of several pixels forming a unit which has to be processed by an algorithm. In MDSDF, no virtual token size is given, but calculated indirectly (see Section 3.2.1). As we will see later on, such a definition for WSDF would lead to a *non-deterministic* model.

The aim of decomposing virtual tokens into effective ones is to obtain a model allowing for efficient synthesis. If such a decomposition was not possible, a sink actor could only fire after a complete virtual token would be available. Especially for image processing applications, this leads the inefficient buffer management.

The approach chosen here corresponds in a certain way to that of FRDF [OH02]. From this model, efficient implementations for non-overlapping windows concerning the amount of required memory can be synthesized. As a consequence, this should also be possible for WSDF.



(a) WSDF Graph fragment



(b) Combination of effective tokens to virtual tokens

Figure 20: Example for WSDF token production showing the composition of a virtual token $\overrightarrow{v_0}$ composed by effective tokens $\overrightarrow{p_0}$

5.4 WSDF Token Consumption

The innovation of WSDF is to allow for "overlapping" multidimensional tokens, so called windows or hyper-windows.

Definition 5.3 *Hyper-Window of dimension n*

A hyper-window is an n-dimensional mask sliding over some source data material, the so called support*. For each position of the hyper-window, a token with the same size as the window is extracted from the source data material.*

Definition 5.4 *Window*

A window is a hyper-window having two dimensions.

In the rest of the document, we will for convenience only use the term window, even if tokens with more than two dimensions are considered.

Definition 5.5 *Virtual Token Union*

Virtual token unions are defined on each WSDF-edge and represent the support for the sliding window operation. They consist of one or more virtual tokens whose number is defined by the function

$$\overrightarrow{u} : E \to \mathbb{N}^n,\ e \mapsto \overrightarrow{u}(e) =: \overrightarrow{u_e}$$

Furthermore, they can contain an extended border due to border processing.

Definition 5.6 *WSDF Token Consumption*

Each time, a sink actor $snk(e)$ is executed, it consumes a multidimensional token of constant size. The size of the token corresponds to that of the window or hyper-window and is defined by the function $\overrightarrow{c}(e)$:

$$\overrightarrow{c} : E \to \mathbb{N}^n,\ e \mapsto \overrightarrow{c}(e) =: \overrightarrow{c_e}$$

An actor v is executable, if on each edge in $\{e \in E : snk(e) = v\}$ enough data elements are stored, so that to each position of the window $\overrightarrow{c}(e)$ a corresponding data element can be assigned.

In the inner of a virtual token union. the window progression *is defined by the diagonal* sampling matrix $\Delta c(e)$*:*

$$\Delta c : E \to \mathbb{N}^{nxn},\ e \mapsto \begin{bmatrix} c_{1,1}(e) & 0 & \cdots & 0 \\ 0 & c_{2,2}(e) & & \vdots \\ \vdots & & \ddots & \\ 0 & \cdots & & c_{n,n}(e) \end{bmatrix} =: \Delta c_e$$

$\langle \Delta c \cdot \overrightarrow{e_i}, \overrightarrow{e_i} \rangle$ defines the window translation in dimension i. All window components must belong to the same support, i.e. virtual token union. If the sliding window leaves a virtual token union, the sampling process is restarted in the upper left corner of the next virtual token union.

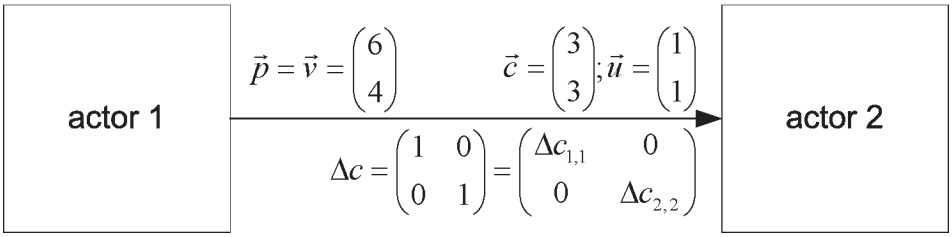


Figure 21: Specification of the sampling grid in WSDF. The window has 3 columns and 3 lines and is moved by one pixel both horizontally and vertically. The sampling with overlapping windows is restricted to a virtual token union consisting of 6 columns and 4 lines. As we only have one edge, the edge indices are omitted.

5.4.1 Window Progression

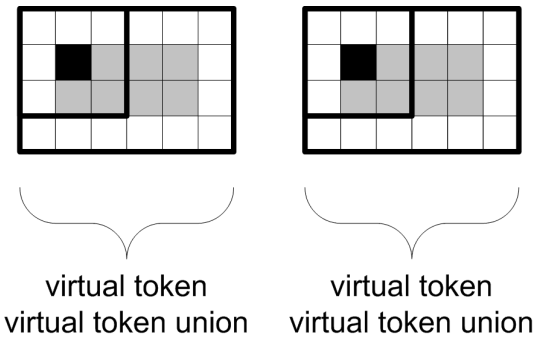
As defined in Definition 5.6, sliding window translation is restricted to one single virtual token union. The latter one can consist of several virtual tokens and an optimal extended border. Figure 21 shows an example WSDF graph with tokens having two dimensions. Actor 1 produces virtual tokens with 6 columns and 4 lines. The virtual token unions formed by actor 2 contain exactly one virtual token without any border extension. The progression of the window is specified by the *sampling matrix* Δc in the following manner:

- $\Delta c \cdot \overrightarrow{e_1} = \begin{pmatrix} \Delta c_{1,1} \\ 0 \end{pmatrix}$ specifies by how many samples the window moves along the first base vector
- $\Delta c \cdot \overrightarrow{e_2} = \begin{pmatrix} 0 \\ \Delta c_{2,2} \end{pmatrix}$ specifies by how many samples the window moves along the second base vector
- More than two dimensions can be treated analogously.

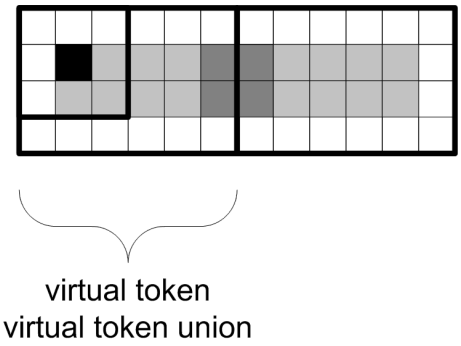
Figure 22(a) shows the token flow corresponding to the graph shown in Figure 21. The black data element represents the center of the window. The gray boxes represent the positions which the center of the window traverses during sampling.

The sampling operation starts in the upper left corner of the virtual token union, so that the window is completely included in the latter one. As a consequence, we need nine new data elements. Then it moves horizontally by $\langle \Delta c \cdot \overrightarrow{e_1}, \overrightarrow{e_1} \rangle = c_{1,1} = 1$ data elements until its right border arrives at the end of the virtual token union. During these invocations $\langle \Delta c \cdot \overrightarrow{e_1}, \overrightarrow{e_1} \rangle \cdot \langle \overrightarrow{c}, \overrightarrow{e_i} \rangle = 1 \cdot 3 = 3$ new data elements are necessary per actor execution. In the following, it moves down by $\langle \Delta c \cdot \overrightarrow{e_2}, \overrightarrow{e_2} \rangle = c_{2,2} = 1$ lines and restarts from the left border of the virtual token union. Now, each actor invocation needs $\langle \Delta c \cdot \overrightarrow{e_2}, \overrightarrow{e_2} \rangle \cdot \langle \overrightarrow{c_1}, \overrightarrow{e_1} \rangle = 1 \cdot 3 = 3$ respectively $\langle \Delta c_2 \cdot \overrightarrow{e_2}, \overrightarrow{e_2} \rangle \cdot \langle \Delta c_1 \cdot \overrightarrow{e_1}, \overrightarrow{e_1} \rangle = 1 \cdot 1 = 1$ new data elements.

The corresponding temporal behavior is shown in Figure 23 in order to clarify the concept of token consumption.



(a) Interpretation 1



(b) Interpretation 2

Figure 22: Possible interpretations of a WSDF model. We suppose, that a virtual token union corresponds exactly to one virtual token (see Figure 21). In part (a), the sampling with an overlapping window is restricted to one single virtual token union as it is demanded by our model. The grey hatched square corresponds to the position of the window's center pixel, when actor 2 fires. Part (b) would be an additional interpretation possibility, if the latter restriction did not exist. If execution of both interpretations was valid, we would have a non-deterministic model.

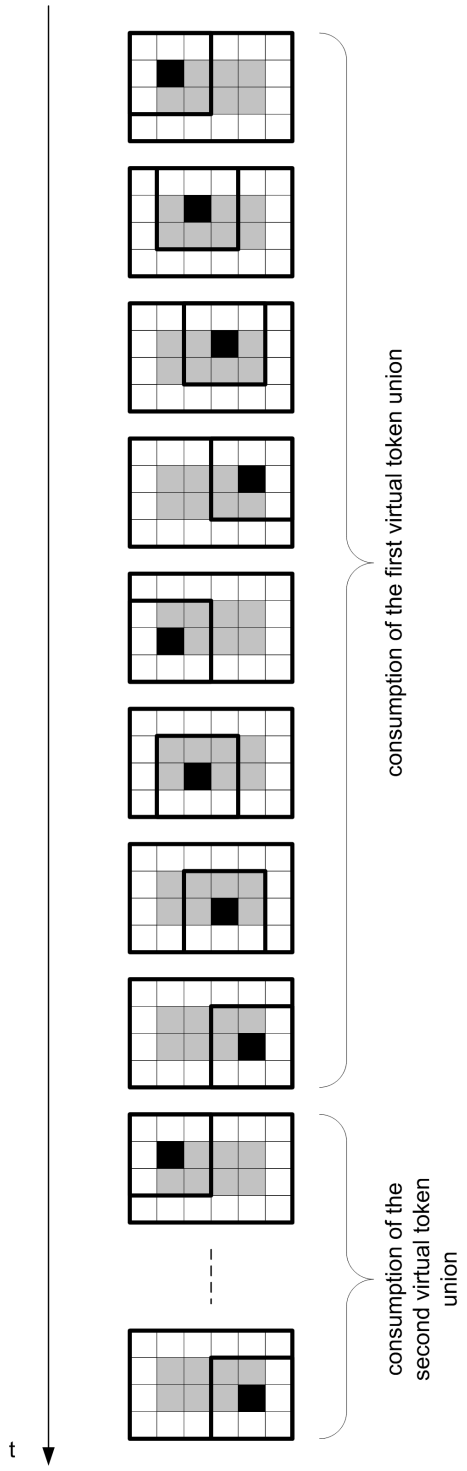


Figure 23: Temporal actor invocation for the example given in Figure 22(a). t is the abstract time. Each element of the shown sequence corresponds to an invocation of actor 2 and shows the corresponding window position.

In the description above, we have supposed a zigzag scanning order. This is only for explication purposes. Like MDSDF, WSDF does not enforce any concrete sampling order.

Definition 5.7 *Definition Initial Invocation in dimension i*

The first execution of a sink actor $snk\,(e)$ in dimension i of a virtual token union is called initial invocation. *It corresponds to the consumption of $\langle \overrightarrow{c_e}, \overrightarrow{e_i} \rangle$ data elements in dimension i. The initial invocation is also called* initial firing.

Example 5.1 *The position of the windows shown in Figure 22(a) correspond to initial invocations in both horizontal and vertical dimension. After moving it one pixel to the right, only in vertical direction an initial invocation is executed.*

5.4.2 The Problem of Ambiguity

In Definition 5.6, we have restricted the sampling with overlapping windows to the inner of one virtual token union. In other words, we have explicitly indicated the grouping of tokens to logical units, where the sampling should take place. For the next virtual token union, we restart the sampling operation.

In original MDSDF without overlapping tokens, this is different (see Section 3.2.1). The firing counts are determined in such a way, that the produced and consumed tokens encompass the same amount of data elements in each dimension.

However, if we proceeded in the same manner for overlapping tokens, we would come out with an *ambiguous* or *non-deterministic* model. This is explained in Figure 22. We base on the WSDF graph shown in Figure 21, however we forget about virtual token unions and the sampling restriction. As a consequence, we have several interpretation possibilities for the given graph:

1. Actor 1 produces one virtual token which is immediately consumed by actor 2. After having finished to consume the first virtual token union, we restart the sampling on the second virtual token union. This results in two times 8 firings of actor 2.

2. In the second interpretation, two virtual token unions are first combined, and then actor 2 starts the sampling operation. As we can see in Figure 22, the sampling with overlapping tokens encompasses more than one virtual token union and leads to two times 10 actor invocations. Although both times actor 1 produces the same amount of data elements, the invocation number of actor 2 differs.

In original MDSDF without overlapping tokens, we do not have such problems. Once the smallest integer repetition counts are determined, we can fire each actor multiple times of this solution without changing the meaning of the model.

The reason why WSDF behaves different is, that at the end of each virtual token union, we want to restart the sampling process as it is shown in Figure 22(a). This behavior is necessary in many image processing applications. In order to avoid the resulting ambiguity, we have to restrict the sampling support. Therefore, we introduced the concept of virtual token unions.

Algorithm 1 Description, how a hyper-window moves for token consumption through a more-dimensional source virtual token union. $\overrightarrow{b^s}$ and $\overrightarrow{b^t}$ model border processing and are explained in Section 5.7. Each time a valid position is found, the function *ExtractWindow* extracts the corresponding data elements.

```
//initialize window_corner_position
for(i=0; i<n; i++)
   window_corner_position[i]=0;
end_virtual_token_union = false;
while (!end_virtual_token_union){
   i = 0; end_line = true;
   while (end_line && !end_virtual_token_union) {
      window_corner_position[i] += ⟨Δc_e · e_i, e_i⟩;
      if (window_corner_position[i] + ⟨c_e, e_i⟩ >= ⟨v_e, e_i⟩ ·
⟨u_e, e_i⟩ + ⟨b^s + b^t, e_i⟩){
         //end of virtual token union line
         end_line = true;
         window_corner_position[i] = 0;
         i += 1;
         if (i >= n)
            end_virtual_token_union = true;
      }else{
         end_line = false;
         ExtractWindow(window_corner_position);
      }
   }
}
```

5.4.3 More Than Two Dimensions

In Section 5.4.1, our explanation has been restricted to two dimensions, because this situation can be easily pictured.

However, the extension to more than two dimensions is straightforward. Algorithm 1 shows the behavior for an arbitrary number of token dimensions n supposing once again zigzag scanning. In order to simplify the code, we do not determine the position of the window center, but of its upper left corner.

5.5 Interpretation of Data Elements, Virtual Tokens and Virtual Token Unions

The introduction of virtual tokens and virtual token unions serves the purpose to avoid non-determinism. However, the introduction of both virtual tokens and virtual token unions is in

some way redundant.

Motivated by our image processing applications, we nevertheless decided to introduce both of them: Virtual tokens correspond to images, composed of *data elements* called *pixels*. Given an image processing operation in form of an actor, the size of the output image depends from the algorithm and the size of the input image. This is the reason, why we assigned the virtual token size to the output of the corresponding actor.

Some algorithms however might need to consume more than one image as input. For instance, we might want to combine two consecutive images in order to calculate their difference. This is the reason for the introduction of virtual token unions. In principle, it would have been sufficient to assign the virtual token size to the sink actor. Then, however, the latter one would have decided about the data interpretation of the source actor which is not very intuitive as solution. Furthermore, the fact, that a sink actor needs more than one image as input would have been less intuitive.

In the following, we suppose that the virtual token unions have the same size as the virtual tokens, if $\overrightarrow{u}(e)$ is not specified in the WSDF graph.

5.6 WSDF Delay Elements

The WSDF model uses the same interpretation of delay elements as the MDSDF model (see section 3.2.2).

Definition 5.8 *WSDF Delay Element*

Each edge e of a WSDF graph G can have a positive, multidimensional delay $\overrightarrow{d}(e)$

$$\overrightarrow{d} : E \to \mathbb{N}_0^n,\ e \mapsto \overrightarrow{d}(e) := \overrightarrow{d_e}$$

$\langle \overrightarrow{d}(e), \overrightarrow{e_i} \rangle$ *defines the number of initial hyper planes orthogonal to* $\overrightarrow{e_i}$.

5.7 Modeling of Border Processing

The application of a sliding window algorithm without performing any border processing leads to output images which do not have the same size than the input images. In most cases, this is not desired. In order to remedy the situation, the input image is virtually extended by a border as shown in Figure 24.

In WSDF, being n the number of dimensions, border processing is modeled by the two n-dimensional vectors $\overrightarrow{b^s}$ and $\overrightarrow{b^t}$:

$$\overrightarrow{b^{s,t}} : E \to \mathbb{Z}^n,\ e \mapsto \overrightarrow{b^{s,t}}(e) =: \overrightarrow{b_e^{s,t}}$$
$$\langle \overrightarrow{u}, \overrightarrow{e_i} \rangle \cdot \langle \overrightarrow{v}, \overrightarrow{e_i} \rangle + \langle \overrightarrow{b^s}, \overrightarrow{e_i} \rangle + \langle \overrightarrow{b^t}, \overrightarrow{e_i} \rangle > 0$$

Their meaning is exemplified for $n = 2$ in Figure 25.

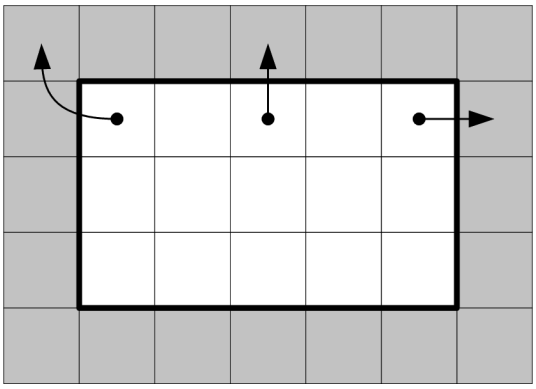


Figure 24: Virtual extension of the input image. The latter one is represented by a thick black border, the hatched squares correspond to the extended pixels. The arrows illustrate border processing by symmetric image extension.

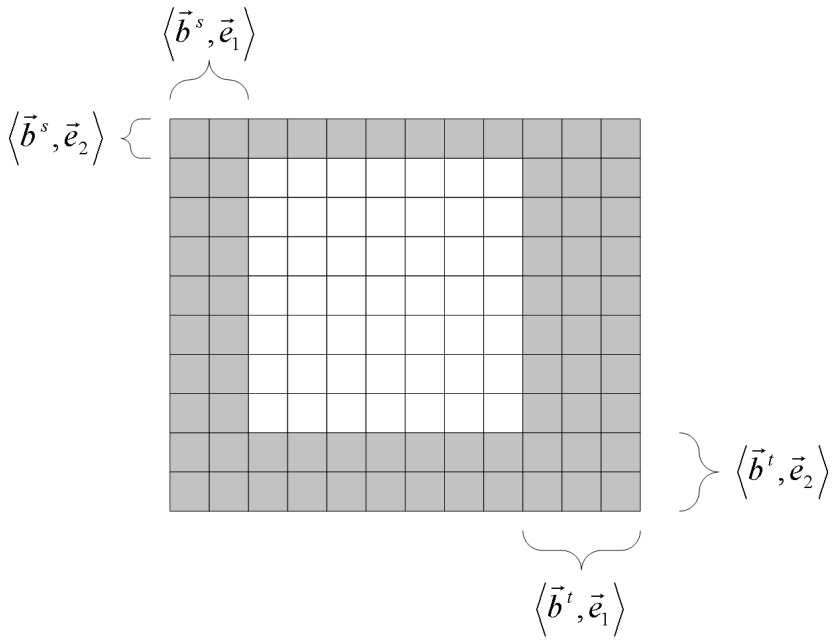


Figure 25: Meaning of the border processing parameters. The grey hatched squares represent the pixels added by the border processing operator to the virtual token union. The pixels belonging to virtual tokens produced by the source actor of the edge are represented by white squares.

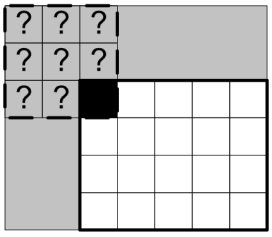


Figure 26: Symmetric border extension for asymmetric windows. The black square represents the center pixel of the window which scans all input image pixels. The input image is framed by a thick black line, the window by a dashed one.

$\langle \overrightarrow{b^s}, \overrightarrow{e_i} \rangle$ specifies, by how many lines respectively columns the beginning of the virtual token union is extended, whereas $\langle \overrightarrow{b^t}, \overrightarrow{e_i} \rangle$ is associated to the end of the virtual token union. The data elements belonging to the extended borders are supposed to be set to a constant and identical value.

For more than two dimensions, the interpretation is straightforward. Notation is the most difficult. For each virtual token union, there exist two borders which are orthogonal to $\overrightarrow{e_i}$, $1 \leq i \leq n$. $\langle \overrightarrow{b^s}, \overrightarrow{e_i} \rangle$ specifies, how many hyper-planes orthogonal to $\overrightarrow{e_i}$ are added to the border having the smaller coordinate. $\langle \overrightarrow{b^t}, \overrightarrow{e_i} \rangle$ has the same meaning for the border having the larger coordinate value in direction $\overrightarrow{e_i}$.

The window propagation in such an extended virtual token union is exactly the same as explained in Section 5.4.1. Only the number of actor invocation changes due to the extended support. However, the window translation does not care, if an underlying pixel belongs to a virtual token or an extended border.

5.7.1 Modeling of Symmetric Border Extension

The border processing operator introduced above assumes, that all border data elements are set to the same value. Nevertheless also symmetric border extension can be represented by such a notation. We suppose, that it is in the responsibility of the sink actor to recognize, that a pixel belongs to the extended border. In such a case, the latter one can be replaced by the desired value, for instance obtained by mirroring the image at the borders.

5.7.2 Border Processing for Windows with Even Extensions

Whereas the above presented approach does not cause any difficulties for symmetric, odd sized windows, special attention must be payed to non-symmetric windows.

Figure 26 shows a corresponding example. If the sink actor receives the data elements belonging to the dashed framed window, it cannot replace the border pixels by the symmetric values, because they are not already known.

In order to solve this problem, the window can be extended to a symmetric 5x5 one. Then it is the task of the actor to determine those pixels which are used to calculate the output. How-

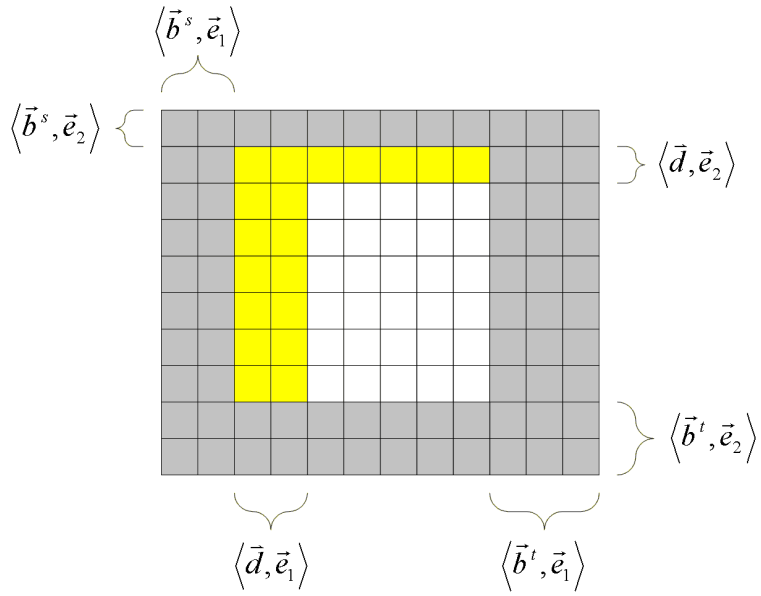


Figure 27: Interpretation of the border processing parameters in combination with a delay operator. As it can be seen by comparison with Figure 25, the meaning of the border processing operator has not changed by the additional delay operator. The latter one defines, how many initial lines and columns of the non-extended virtual token union are already available.

ever, this approach leads to inefficient buffer management, because the larger the window, the more lines must be buffered. Fortunately asymmetric windows combined with symmetric border extension is not very frequent. Consequently, as WSDF in is current version is already quite complex, a further extension enhancing complexity seems not to be worth.

5.7.3 Interdependency between Border Processing Operator and Delay Operator

If an WSDF edge contains a delay operator and a border processing operator, then their effect is defined in the following manner:

- A delay operator represents the number of initially available token hyper-planes. The border processing operator however modifies each virtual token union.
- The delay operator specifies always the number of already available virtual token data elements. It is not influenced by the presence of a border processing operator.

With this definition, the order of the border processing operator and the delay operator does not play any role. Figure 27 shows an example for the meaning of the border processing operator in combination with a delay operator.

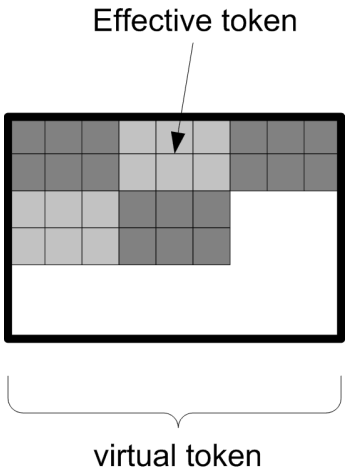


Figure 28: Illustration of the virtual token fill level. The effective tokens consist of 3 lines and two columns. Nine effective tokens together form a virtual token.

5.7.4 Negative Border Processing Values

Most of the border processing algorithms need the extension of the input virtual token union. However, in some cases, pixels have to be eliminated. This for instance is the case for lifting based wavelet transform, where the highpass band can have less pixels than the lowpass one, although they are calculated on the same input image.

Such a scenario can be easily supported by allowing also values of $\langle \overrightarrow{b^s}, \overrightarrow{e_i} \rangle$ and $\langle \overrightarrow{b^t}, \overrightarrow{e_i} \rangle$ being smaller zero. However, the following conditions must be assured:

$$\langle \overrightarrow{u}, \overrightarrow{e_i} \rangle \cdot \langle \overrightarrow{v}, \overrightarrow{e_i} \rangle + \langle \overrightarrow{b^s}, \overrightarrow{e_i} \rangle + \langle \overrightarrow{b^t}, \overrightarrow{e_i} \rangle > 0$$

5.8 State of a WSDF Graph

As for SDF, we look for periodic schedules which do not change the net graph state (see Definition 3.4). A WSDF graph, however, has a cyclo-static behavior. For the initial actor invocation, the number of necessary data elements might be different from the following executions. As a consequence, we define:

Definition 5.9 *Fill Level of a Virtual Token*

Both virtual *and* effective tokens *are arrays of data elements. A virtual token is composed of several effective tokens. The* fill level of a virtual token *is the fraction of already assigned data elements and the overall number of data elements.*

Example 5.2 *Fill Level of a Virtual Token*

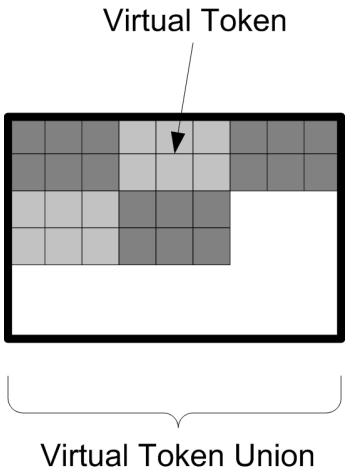


Figure 29: Illustration of the virtual token union fill level. The virtual tokens consist of $2 \cdot 3$ data elements. Nine virtual tokens form one virtual token union.

Figure 28 shows a virtual token which is composed of effective tokens consisting of three columns and two lines. The fill level of the virtual token can be calculated by

$$l_v = \frac{2 \cdot 9 + 2 \cdot 6}{6 \cdot 9} = \frac{30}{54} = \frac{5}{9}$$

Definition 5.10 *Fill Level of a Virtual Token Union*

Virtual token unions are composed of several virtual tokens. The fill level of a virtual token union *is the fraction between the already available virtual tokens and their overall number.*

Example 5.3 *Fill Level of a Virtual Token Union*

Figure 29 shows a virtual token union which is composed by nine virtual tokens. The virtual token union fill level can be calculated by

$$l_u = \frac{5}{9}$$

Whereas the fill level of a virtual token is based on the number of data elements, the corresponding value for the virtual token union is directly derived from the number of contained virtual tokens. The reason for this distinction is, that a virtual token can be composed of a fractional number of effective tokens. A virtual token union however is always composed of an integer number of virtual tokens.

Definition 5.11 *WSDF-Graph State*

The state *of a WSDF graph consists of four different parts:*

1. *The number of data elements stored on the edge buffers*
2. *The fill level of the virtual tokens*

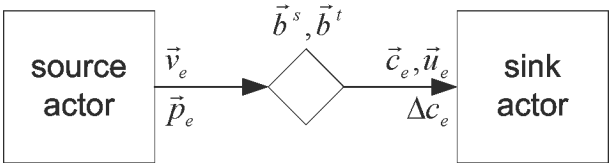


Figure 30: Single edge WSDF Graph

3. *The fill level of the virtual token unions*
4. *The relative positions of the sampling windows in the virtual token unions including extended borders*

In other words: In order to return to the same state, the sampling windows of all sink actors must return to the same position of the input virtual token union, from which they started. Furthermore, the fill level of the occurring virtual tokens and virtual token unions must be the same. Whereas border processing indirectly influences the WSDF-graph state, initial tokens do not have any impact.

5.9 WSDF Balance Equation

If we consider the WSDF behavior for token consumption and production, we recognize a cyclo-static behavior. For the first invocation of an actor in the inner of a virtual token union, the token consumption differs from the successive invocations. As furthermore the firing for two different dimensions i_1 and i_2, $i_1 \neq i_2$ are independent of each other, we expect the balance equation to be similar to that one of the CSDF model.

Figure 30 shows two WSDF actors connected by one edge. Based on this figure, we will derive a balance equation which guarantees, that a WSDF graph returns to its initial state.

Corollary 5.1 *Given a sink actor $snk\,(e)$ of edge e. Then, the number of invocations on edge e for one single input virtual token union and for dimension i is given by*

$$\langle \overrightarrow{r_{vtu}}\,(e)\,,\,\overrightarrow{e_i}\rangle = \frac{\langle \overrightarrow{u_e},\overrightarrow{e_i}\rangle \cdot \langle \overrightarrow{v_e},\overrightarrow{e_i}\rangle + \langle \overrightarrow{b_e^s}+\overrightarrow{b_e^t},\overrightarrow{e_i}\rangle - \langle \overrightarrow{c_e},\overrightarrow{e_i}\rangle}{\langle \Delta c_e \cdot \overrightarrow{e_i},\overrightarrow{e_i}\rangle} + 1 \tag{6}$$

Proof 5.1 *For the initial invocation in dimension i, we need $\langle \overrightarrow{c_e},\overrightarrow{e_i}\rangle$ new data-elements. For each consecutive firing, we only need $\langle \Delta c_e \cdot \overrightarrow{e_i},\overrightarrow{e_i}\rangle$ new data elements, because tokens can overlap. As a consequence, supposing one virtual token union, the number of consumed tokens can be calculated to*

$$\underbrace{\langle \overrightarrow{c_e},\overrightarrow{e_i}\rangle + \langle \Delta c_e \cdot \overrightarrow{e_i},\overrightarrow{e_i}\rangle \cdot (\langle \overrightarrow{r_{vtu}}\,(e)\,,\overrightarrow{e_i}\rangle - 1)}_{\text{consumed tokens in dimension i}} = \underbrace{\langle \overrightarrow{v_e},\overrightarrow{e_i}\rangle \cdot \langle \overrightarrow{u_e},\overrightarrow{e_i}\rangle + \langle \overrightarrow{b^s}+\overrightarrow{b^t},\overrightarrow{e_i}\rangle}_{\text{size of virtual token union in dimension i, including extended borders}}$$

Corollary 5.2 *Sampling with a sliding window is restricted to the inner of a virtual token union, if and only if*

$$\forall 1 \leq i \leq n: \ \langle \overrightarrow{r_{vtu}}(e), \overrightarrow{e_i} \rangle \in \mathbb{N} \tag{7}$$

Proof 5.2 *Necessary condition*

A sink actor has to fire at least one time per virtual token union. For the first sink actor invocation, $\langle \overrightarrow{c_e}, \overrightarrow{e_i} \rangle$ *data elements in dimension* i *are necessary. As a consequence,* $\langle \overrightarrow{u_e}, \overrightarrow{e_i} \rangle \cdot \langle \overrightarrow{v_e}, \overrightarrow{e_i} \rangle - \langle \overrightarrow{c_e}, \overrightarrow{e_i} \rangle \geq 0$*. If* $\langle \overrightarrow{r_{vtu}}(e), \overrightarrow{e_i} \rangle \notin \mathbb{Z}$ *, the initial actor invocation could not take place at the border of a virtual token union, because either there would "stay" some data elements from the previous virtual token union or some data elements of the new virtual token union would have already be consumed during the actor invocations belonging to the previous virtual token union.*

Proof 5.3 *Sufficient condition*

If $\langle \overrightarrow{r_{vtu}}(e), \overrightarrow{e_i} \rangle \in \mathbb{N}$*, the actor* $snk(e)$ *fires at least one time. Furthermore, it has consumed exactly the number of data elements belonging to one single virtual token union. As a consequence, we can restart sampling operation with the next new virtual token union.*

Example 5.4 *For the graph given in Figure 21, the vector* $\overrightarrow{r_{vtu}}$ *can be calculated to*

$$\overrightarrow{r_{vtu}} = \begin{pmatrix} \frac{1 \cdot 6 - 3}{1} + 1 \\ \frac{1 \cdot 4 - 3}{1} + 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \end{pmatrix}$$

This corresponds exactly to what is shown in Figure 22(a).

Definition 5.12 *Valid WSDF graph*

A WSDF graph is called valid*, if and only if*

$$\forall e \in E \, and \, \forall 1 \leq i \leq n : \langle \overrightarrow{r_{vtu}}(e), \overrightarrow{e_i} \rangle \in \mathbb{N}$$

If an actor is the sink of multiple edges, then we can calculate a minimal period $L_i(v)$ representing the minimal number of invocations of actor v in dimension i after which the relative positions of the sampling windows in the virtual token unions of all edges are returned to their initial state.

Definition 5.13 *Actor Period*

The period of an actor v *is the number of invocations, after which the relative positions of the sampling windows in the virtual token unions of all input edges are returned to their initial state.*

Corollary 5.3 *Given an actor* $v \in V$*. Then the minimal actor period* $\overrightarrow{L}(v)$ *can be calculated by*

$$\forall 1 \leq i \leq n: \ L_i(v) := \langle \overrightarrow{L}(v), \overrightarrow{e_i} \rangle = scm_{e \in E, \, snk(e) = v} \left(\langle \overrightarrow{r_{vtu}}(e), \overrightarrow{e_i} \rangle \right) \tag{8}$$

For an actor which is not sink of any edge, this yields to

$$\forall 1 \leq i \leq n: \ L_i(v) = 1$$

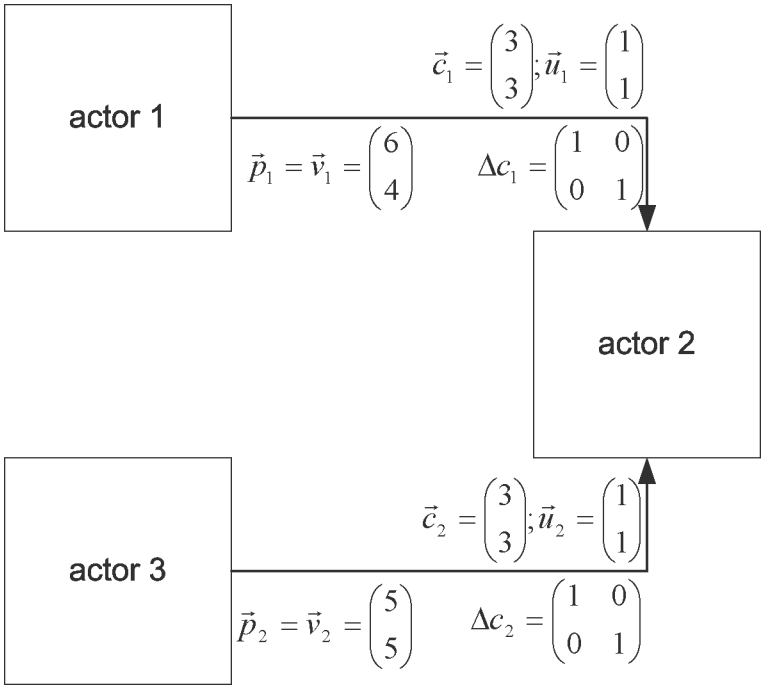


Figure 31: Example WSDF-graph for illustration of the actor period concept. Both actor 1 and actor 3 are pure source actors. Consequently, $\overrightarrow{L}(v_1) = \overrightarrow{L}(v_3) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$. Actor 2 is the sink of two edges e_1 and e_2. With the help of Equation (8), we get $\overrightarrow{L}(v_2) = \begin{pmatrix} 12 \\ 6 \end{pmatrix}$

Proof 5.4 *For each edge, the relative window position in the virtual token union returns to its original position after consumption of a complete virtual token. As a consequence, the number of actor invocations must be a multiple of* $\langle \overrightarrow{r_{vtu}}(e), \overrightarrow{e_i} \rangle$ *in order to return the window position to the original one. For actors not being the sink of any edge, we set the minimal actor period to one, because the token consumption is constant for each actor invocation. This is expressed by the fact, that*

$$\langle \overrightarrow{L}(v), \overrightarrow{e_i} \rangle = scm_{e \in E,\, snk(e)=v} \left(\langle \overrightarrow{r_{vtu}}(e), \overrightarrow{e_i} \rangle \right) = scm(\emptyset) = 1$$

Example 5.5 *Figure 31 shows an example graph consisting of three actors. Actor 1 and actor 3 are pure source actor. As a consequence, we get* $\overrightarrow{L}(v_1) = \overrightarrow{L}(v_3) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$. *Actor 2 is the sink of two edges* e_1*and* e_2 *with the following repetition count for one single virtual token union*

$$\overrightarrow{r_{vtu}}(e_1) = \begin{pmatrix} 4 \\ 2 \end{pmatrix}, \overrightarrow{r_{vtu}}(e_2) = \begin{pmatrix} \frac{1 \cdot 5 - 3}{1} + 1 \\ \frac{1 \cdot 5 - 3}{1} + 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 3 \end{pmatrix}$$

By help of equation (8), we can derive for the actor period

$$\overrightarrow{L}(v_2) = \begin{pmatrix} scm(4,3) \\ scm(2,3) \end{pmatrix} = \begin{pmatrix} 12 \\ 6 \end{pmatrix}$$

.

Theorem 5.4 *WSDF Graph Balance Equation*
Given a valid WSDF-Graph $G = \left(V, E, \overrightarrow{p}, \overrightarrow{v}, \overrightarrow{c}, \Delta c, \overrightarrow{u}, \overrightarrow{d}, \overrightarrow{b^s}, \overrightarrow{b^t}\right)$*, so that* $\forall e \in E \wedge \forall 1 \leq i \leq n : \langle \overrightarrow{r_{vtu}}(e), \overrightarrow{e_i} \rangle \in \mathbb{N}$*. Then the number of actor invocations in dimension* i *in order to return the WSDF graph to its initial state can be calculated by*

$$\overrightarrow{r_i} = M_i \cdot \overrightarrow{q_i}, \text{ with } M_i = \begin{bmatrix} L_i(v_1) & 0 & \cdots & 0 \\ 0 & L_i(v_2) & & \\ \vdots & & \ddots & \\ 0 & & & L_i\left(v_{|V|}\right) \end{bmatrix} \tag{9}$$

Both $\overrightarrow{r_i}$ *and* $\overrightarrow{q_i}$ *are strictly positive integer vectors.*
$\overrightarrow{r_i}(v_j) := \langle \overrightarrow{r_i}, \overrightarrow{e_j} \rangle$ *is the number of invocations of actor* v_j *in dimension* i*.*
$\overrightarrow{q_i}(v_j) := \langle \overrightarrow{q_i}, \overrightarrow{e_j} \rangle$ *defines, how often the minimal period of actor* v_j *in dimension* i *is executed and can be calculated by*

$$\underbrace{\begin{bmatrix} \Gamma_{1,1,i} & \cdots & \Gamma_{1,v,i} & \cdots & \Gamma_{1,|V|,i} \\ \vdots & & \vdots & & \vdots \\ \Gamma_{e,1,i} & \cdots & \Gamma_{e,v,i} & \cdots & \Gamma_{e,|V|,i} \\ \vdots & & \vdots & & \vdots \\ \Gamma_{|E|,1,i} & \cdots & \Gamma_{|E|,v,i} & \cdots & \Gamma_{|E|,|V|,i} \end{bmatrix}}_{\Gamma_i} \cdot \overrightarrow{q_i} = \overrightarrow{0} \tag{10}$$

with

$$\Gamma_{e,v,i} = \begin{cases} L_i(v) \cdot \langle \overrightarrow{p_e}, \overrightarrow{e_i} \rangle & \text{if } v = src(e) \\ 0 & \text{otherwise} \end{cases} - \begin{cases} \frac{L_i(v)}{\langle \overrightarrow{r_{vtu}}(e), \overrightarrow{e_i} \rangle} \cdot (\langle \overrightarrow{v_e}, \overrightarrow{e_i} \rangle \cdot \langle \overrightarrow{u_e}, \overrightarrow{e_i} \rangle) & \text{if } v = snk(e) \\ 0 & \text{otherwise} \end{cases}$$

Proof 5.5 *For each actor* $v \in V$*, the number of invocations must be a multiple of its minimal period. Otherwise, some windows would not return to their initial position and the graph state would change. This condition is assured by Equation (9).*

1. $v = snk(e)$*:*
   The execution of one minimal period of actor v *corresponds to* $L_i(v)$ *actor invocations. As the consumption of one virtual token union needs* $\langle \overrightarrow{r_{vtu}}(e), \overrightarrow{e_i} \rangle$ *actor firings,* $\frac{L_i(v)}{\langle \overrightarrow{r_{vtu}}(e), \overrightarrow{e_i} \rangle} \in \mathbb{N}$ *virtual token unions are consumed. For each virtual token union,*

   $$\langle \overrightarrow{c_e}, \overrightarrow{e_i} \rangle + (\langle \overrightarrow{r_{vtu}}(e), \overrightarrow{e_i} \rangle - 1) \cdot \langle \Delta c_e \cdot \overrightarrow{e_i}, \overrightarrow{e_i} \rangle = \langle \overrightarrow{u}, \overrightarrow{e_i} \rangle \cdot \langle \overrightarrow{v}, \overrightarrow{e_i} \rangle + \langle \overrightarrow{b^s} + \overrightarrow{b^t}, \overrightarrow{e_i} \rangle$$

   data elements are consumed (see Corollary 5.1). However, only $\langle \overrightarrow{u_e}, \overrightarrow{e_i} \rangle \cdot \langle \overrightarrow{v_e}, \overrightarrow{e_i} \rangle$ *of them have been produced by the source actor, the others are introduced by the border processing operator.*

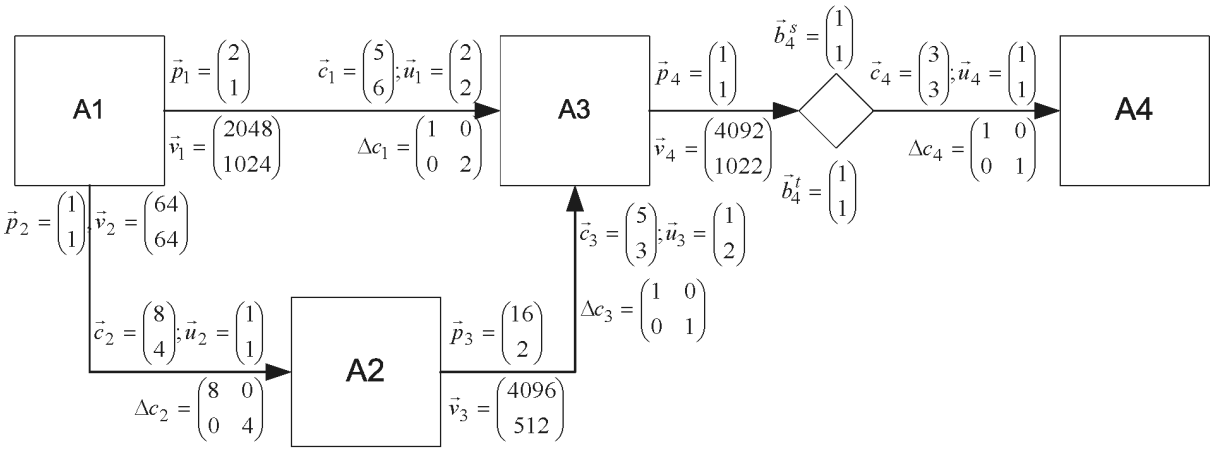


Figure 32: Example WSDF Graph

2. $v = src(e)$:
   The execution of one minimal period of actor v corresponds to $L_i(v)$ actor invocations. For each source actor invocation, $\langle \overrightarrow{p_e}, \overrightarrow{e_i} \rangle$ data elements are produced.

If Equation (10) holds, then the number of produced and consumed tokens is identical. As we execute each actor an integer multiple times of its minimal actor period, we are sure, that only complete virtual token unions are consumed. Consequently, the same is valid for production and the WSDF graph returns to its initial state.

As the topology matrix in Equation (10) has exactly the same properties as for SDF graphs, we can derive from [LM87] that for a connected graph there exists a strictly positive and integer vector $\overrightarrow{q_i}$, if and only if

$$rank(\Gamma_i) = |V| - 1$$

Consequently, also $\overrightarrow{r_i}$ is integer and strictly positive.

5.9.1 Example

Figure 32 shows an example WSDF graph. The graph behavior is borrowed from image processing, however generalized in some aspects in order to show all elements of the WSDF model. Actor 1 produces two output images. The first one has a size of 2048*1024 pixels. The size of the second image shall be identical, however it is cut in blocks of 64x64 pixels each. Actor 2 takes these blocks, performs a down-sampling in vertical direction and an up-sampling in horizontal direction and combines the resulting blocks to an image with the size 4096*512 pixels. Actor 3 consumes 2x2 images of actor 1 and down-samples them in vertical direction. Furthermore, it consumes two images of actor 2. Both the images from actor 1 and actor 2 are processed with a sliding window of size 5x6, respectively 5x3. The result is passed to actor 4 which performs a sliding window operation with border extension.

Edge	1	2	3	4
$\overrightarrow{r_{vtu}}(e)$	$\begin{pmatrix} 4092 \\ 1022 \end{pmatrix}$	$\begin{pmatrix} 8 \\ 16 \end{pmatrix}$	$\begin{pmatrix} 4092 \\ 1022 \end{pmatrix}$	$\begin{pmatrix} 4092 \\ 1022 \end{pmatrix}$

Table 2: Repetition counts for consumption of a single virtual token union, based in the graph shown in Figure 32

Actor	1	2	3	4
$\overrightarrow{L}(v_j)$	$\begin{pmatrix} 1 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 8 \\ 16 \end{pmatrix}$	$\begin{pmatrix} 4092 \\ 1022 \end{pmatrix}$	$\begin{pmatrix} 4092 \\ 1022 \end{pmatrix}$

Table 3: Minimal actor periods for the example shown in Figure 32

Actor Periods First of all, we have to calculate the minimal periods of each actor. The corresponding results are shown in Tables 2 and 3.

The topology matrices for both dimensions allow for the calculation of the vectors $\overrightarrow{q_i}$:

$$\underbrace{\begin{bmatrix} 2 & 0 & -4096 & 0 \\ 1 & -64 & 0 & 0 \\ 0 & 128 & -4096 & 0 \\ 0 & 0 & 4092 & -4092 \end{bmatrix}}_{\Gamma_1} \cdot \begin{pmatrix} q_{1,1} \\ q_{1,2} \\ q_{1,3} \\ q_{1,4} \end{pmatrix} = \overrightarrow{0}$$

$$\underbrace{\begin{bmatrix} 1 & 0 & -2048 & 0 \\ 1 & -64 & 0 & 0 \\ 0 & 32 & -1024 & 0 \\ 0 & 0 & 1022 & -1022 \end{bmatrix}}_{\Gamma_2} \cdot \begin{pmatrix} q_{2,1} \\ q_{2,2} \\ q_{2,3} \\ q_{2,4} \end{pmatrix} = \overrightarrow{0}$$

It is easy to verify, that

$$rank(\Gamma_1) = rank(\Gamma_2) = 3 = |V|$$

The minimal number of actor invocations in multiple of actor periods can be calculated to

$$\overrightarrow{q_1} = \begin{pmatrix} q_{1,1} \\ q_{1,2} \\ q_{1,3} \\ q_{1,4} \end{pmatrix} = \begin{pmatrix} 2048 \\ 32 \\ 1 \\ 1 \end{pmatrix}, \overrightarrow{q_2} = \begin{pmatrix} q_{2,1} \\ q_{2,2} \\ q_{2,3} \\ q_{2,4} \end{pmatrix} = \begin{pmatrix} 2048 \\ 32 \\ 1 \\ 1 \end{pmatrix}$$

As a consequence, the repetition vectors amount

$$\overrightarrow{r_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 \\ 0 & 0 & 4092 & 0 \\ 0 & 0 & 0 & 4092 \end{bmatrix} \cdot \overrightarrow{q_1} = \begin{pmatrix} 2048 \\ 256 \\ 4092 \\ 4092 \end{pmatrix}$$

$$\overrightarrow{r_2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 16 & 0 & 0 \\ 0 & 0 & 1022 & 0 \\ 0 & 0 & 0 & 1022 \end{bmatrix} \cdot \overrightarrow{q_2} = \begin{pmatrix} 2048 \\ 512 \\ 1022 \\ 1022 \end{pmatrix}$$

5.10 Compatibility with MDSDF

WSDF is an extension of the MDSDF model. As a consequence, it comprises all capabilities of MDSDF. If neither the sampling matrix nor the virtual token or virtual token union size is specified, we suppose MDSDF behavior in order to stay compatible with the MDSDF model. This can be realized by the following assignment:

- $\Delta c\,(e) = diag\,(\overrightarrow{c}\,(e)) := \begin{pmatrix} \langle \overrightarrow{c}\,(e)\,, \overrightarrow{e_1} \rangle & 0 \\ 0 & \langle \overrightarrow{c}\,(e)\,, \overrightarrow{e_2} \rangle \end{pmatrix}$
- $\overrightarrow{u}\,(e) = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$
- $\overrightarrow{v}\,(e) = \overrightarrow{c}\,(e)$
- $\overrightarrow{b^s}\,(e) = \overrightarrow{b^t}\,(e) = \overrightarrow{0}$

This leads to the following relation:

$$\langle \Delta c_e \cdot \overrightarrow{e_i}, \overrightarrow{e_i} \rangle = \langle \overrightarrow{c_e}, \overrightarrow{e_i} \rangle$$

5.10.1 Solution of Balance Equation

Given a WSDF graph $G = \left(V, E, \overrightarrow{p}, \overrightarrow{v}, \overrightarrow{c}, \Delta c, \overrightarrow{u}, \overrightarrow{d}, \overrightarrow{b^s}, \overrightarrow{b^t}\right)$ which shall have MDSDF behavior. Using the above assignments, we obtain from Equation (6):

$$\langle \overrightarrow{r_{vtu}}\,(e)\,, \overrightarrow{e_i} \rangle = 1$$

As a consequence, the period of all actors is one:

$$\forall v \in V, \forall 1 \leq i \leq n : \langle \overrightarrow{L}\,(v)\,, \overrightarrow{e_i} \rangle = 1$$

From Theorem 5.4, we get for the graph balance equation

$$\Gamma_i \cdot \overrightarrow{r_i} = \overrightarrow{0}$$

with

$$\Gamma_{e,a,i} = \begin{cases} \langle \overrightarrow{p_e}, \overrightarrow{e_i} \rangle & \text{if } v = src\,(e) \\ 0 & \text{otherwise} \end{cases} - \begin{cases} \langle \overrightarrow{c_e}, \overrightarrow{e_i} \rangle & \text{if } v = snk\,(e) \\ 0 & \text{otherwise} \end{cases}$$

This corresponds exactly to what is given in [ML02].

6 Conclusion and Future Work

In this report, we analyzed, how well currently known static models of computation represent image processing algorithms based on sliding windows. We found out, that it is difficult to obtain a complete and realistic representation. As a consequence, we developed a new model of computation called *Windowed Synchronous Data Flow*. Its major innovation is the support for overlapping, multi dimensional tokens. As WSDF is static, it offers the possibilities for various analyzes. We derived the balance equation in order to check for limited token accumulation on a single edge.

WSDF in its current state can model many important image processing algorithms as edge detection, correlation, downsampling, block building or simple morphologic operations. However, due to its static nature, there are some restrictions. Both dynamic algorithms as well as changing window sizes can only be represented in a very restricted way or even not at all. Nevertheless, WSDF in its current state can serve as basis to solve many important questions about scheduling and buffer estimation for a large class of multi-dimensional applications. Promising preliminary results in this domain are already available and they are planed to be detailed soon.

References

[ALP97] Marleen Adé, Rudy Lauwereins, and J. A. Peperstraete. Data memory minimisation for synchronous data flow graphs emulated on dsp-fpga targets. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 64–69, New York, NY, USA, 1997. ACM Press.

[BELP96] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, February 1996.

[Bha99] Bishnupriya Bhattacharya. Parameterized modeling and scheduling for dataflow graphs. Master's thesis, Departement of Electrical and Computer Engineering, University of Maryland, November 1999.

[BML99] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems*, 21(2):151–166, June 1999.

[Buc93] Joseph Tobin Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, UNIVERSITY of CALIFORNIA at BERKELEY, 1993.

[Che94] Michael J. Chen. Developing a multidimensional synchronous dataflow domain in ptolemy. Technical Report UCB/ERL M94/16, Electronics Research Laboratory, University of California, Berkely, June 1994.

[HB01] J. P. Hammes and A. P. W. Böhm. *The SA-C language - Version 1.0*. Colorado State University, June 21 2001.

[ISO] JPEG2000 image coding system.

[KM66] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination and queuing. *SIAM J. Appl. Math.*, no. 14(no. 6):1390–1411, Nov. 1966.

[LM87] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1), January 1987.

[ML02] Praveen K. Murthy and Edward A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, Vol50(7):2064–2079, July 2002.

[OH02] Hyunok Oh and Soonhoi Ha. Fractional rate dataflow model and efficient code synthesis for multimedia applications. *LCTES-SCOPES*, 2002.

[WELP96] Piet Wauters, Marc Engels, Rudy Lauwereins, and J.A. Peperstraete. Cyclo-dynamic dataflow. Internal Report ESAT/ACCA/95/2, Katholieke Universiteit Leuven, ESAT Departement, Kard. Mercierlann 94, B-3001 Heverlee, Belgium, Januray 1996.