Adaptive streaming applications : analysis and implementation models

Zhai, J.T.

Chapter 2

Models-of-Computation (MoC)

THIS chapter is dedicated to different Models of Computation (MoC) that serve as the application specification. In particular, we focus on a process-based MoC, namely Polyhedral Process Networks (PPN), in Section 2.1, and two actor-based MoCs, SDF and CSDF in Sections 2.2.1 and 2.2.2, respectively. The PPN MoC is used as the implementation model in Daedalus^{RT} and it is the input to the solutions proposed in Chapters 3 and 4. The SDF MoC is the input to the solution proposed in Chapter 5. The CSDF MoC is used to perform HRT analysis. In Section 2.3, an overview of the HRT analysis is given to better understand the solutions proposed in Chapters 5 and 7. Throughout this thesis, we use the set of mathematical notations listed in Table 2.1.

Both PPN and (C)SDF MoCs are specified as a graph consisting of vertices and edges. Normally, all vertices denote concurrently executing computation tasks. For (C)SDF, the vertices are called *actors*, whereas the vertices in a PPN are called *processes*. The edges denote FIFOs for data communication between actors/processes. It is possible to compute a safe FIFO size [110, 125] for each edge that guarantees the absence of deadlock in the graph.

2.1 Polyhedral Process Networks (PPN)

An important advantage of adopting the PPN MoC in Daedalus^{RT} is that it can be automatically derived from an input-output equivalent sequential specification with certain restrictions using the PNgen [125] compiler. Thus, the error-prone process of deriving a concurrent model manually can be avoided. Moreover, the ESPAM [96] tool is able to generate final parallel implementation for the PPN MoC in an automated way. Consequently, design productivity can be significantly improved. In the following sub-sections, we first explain the *polytope model*, which

Notation	Meaning
$\mathbb{N}$	the set of natural numbers excluding zero
$\mathbb{Q}$	the set of rational numbers
$\mathbb{Z}$	the set of integer numbers
$\check{x}$	lower bound (minimum) of values x
$\hat{x}$	upper bound (maximum) of values x
lcm	least common multiple
$\lceil x \rceil$	smallest integer that is greater than or equal to x
$\lfloor x \rfloor$	greatest integer that is smaller than or equal to x
$\lvert\mathcal{X}\rvert$	cardinality of a set $\mathcal{X}$
$\vec{x}$	vector x

Table 2.1: Mathematical notations.

we use as the formal representation of the PPN MoC and the P^{3}N MoC developed in Chapter 6. It is followed by an explanation of the sequential specification with restrictions in detail. Then, we introduce the PPN MoC based on the polytope model derived from the sequential specification.

Polytope Model

The *Polytope model* [42] is often used in the compiler domain to represent loop nests, which perfectly match the behavior of streaming applications. The polytope model allows powerful transformation techniques that are used to explore and exploit parallelism in Chapters 3 and 4. It also serves as the foundation of the analysis presented in Chapter 6. This section presents an overview of the polytope model to make this thesis self-contained. A more detailed treatment of the polytope model can be found in [26]. The mathematical background can be found in popular textbooks, such as [105]. Throughout this thesis, the notations related to the polytope mode are listed in Table 2.2.

We start with some fundamental definitions. Assuming a vector $\vec{y} \in \mathbb{R}^n$ and a constant α, $\mathcal{H} = \{\vec{x} \mid \vec{x} \cdot \vec{y}^T \geq \alpha\}$ is called a *closed half-space*. Then, we define a *polyhedron* as follows:

Definition 2.1.1 (Polyhedron)**.** A polyhedron $\mathcal{D}$ is the intersection of a set of finitely many closed half-space, i.e.,

$$\mathcal{D} = \{\vec{x} \in \mathbb{Q}^d \mid A\vec{x} \geq \vec{c}\}, \tag{2.1}$$

where $A \in \mathbb{Z}^{m \times d}$ is a constant matrix and $c \in \mathbb{Z}^m$ is a constant vector.

$\mathcal{D}$	a polyhedron
$\mathcal{D}(\vec{p})$	a parametric polyhedron
$\bar{\mathcal{D}}$	a polytope
$\bar{\mathcal{D}}(\vec{p})$	a parametric polytope
$\lvert\bar{\mathcal{D}}\rvert$	cardinality of polytope
R	dependence relation
ran$R$	range of a dependence relation
dom$R$	domain of a dependence relation

Table 2.2: Polyhedral notations.

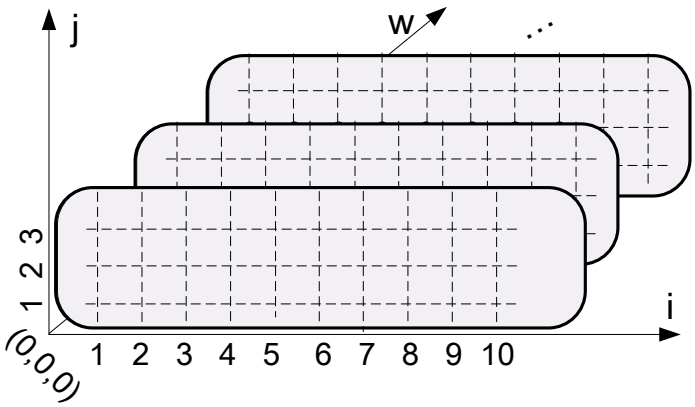


Figure 2.1: A polyhedron.

Definition 2.1.2 (Polytope)**.** A polytope $\bar{\mathcal{D}}$ is a bounded polyhedron.

Consider for instance a polyhedron defined as follows:

$$\mathcal{D} = \left\{ (w,i,j) \in \mathbb{Q}^3 \mid \begin{bmatrix} 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} w \\ i \\ j \end{bmatrix} \geq \begin{bmatrix} 1 \\ -10 \\ 1 \\ -3 \\ 0 \end{bmatrix} \right\},$$
$$= \{(w,i,j) \in \mathbb{Q}^3 \mid w \geq 0 \wedge 1 \leq i \leq 10 \wedge 1 \leq j \leq 3\}. \tag{2.2}$$

The polyhedron is illustrated in Figure 2.1 using grey boxes. We can see that the

polyhedron is unbounded along w-dimension. If we consider any w equal to a constant c, we obtain a polytope $\bar{\mathcal{D}}_w$ as:

$$\bar{\mathcal{D}} = \left\{ (w,i,j) \in \mathbb{Q}^3 \mid \begin{bmatrix} 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} w \\ i \\ j \end{bmatrix} \geq \begin{bmatrix} 1 \\ -10 \\ 1 \\ -3 \\ c \\ -c \end{bmatrix} \right\},$$
$$= \{(w,i,j) \in \mathbb{Q}^3 \mid w = c \wedge 1 \leq i \leq 10 \wedge 1 \leq j \leq 3\}.$$

We can see that the initial $\mathcal{D}$ in Equation (2.2) is now bounded on the w dimension. More specifically, $\bar{\mathcal{D}}$ can be considered as a "plane" spread along i and j axes shown in Figure 2.1.

In Chapter 6, we use the concept of parametric polyhedron to represent adaptive streaming applications.

Definition 2.1.3 (Parametric Polyhedron)**.** A parametric polyhedron $\mathcal{D}(\vec{p})$ is a polyhedron $\mathcal{D}$ affinely depending on a parameter vector $\vec{p} \in \mathbb{Q}^n$, i.e.,

$$\mathcal{D}(\vec{p}) = \{\vec{x} \in \mathbb{Q}^d \mid A \cdot \vec{x} \geq B \cdot \vec{p} + \vec{b}\}, \tag{2.3}$$

where $\vec{p}$ is bounded by a polytope $\bar{\mathcal{D}}_{\vec{p}} = \{\vec{p} \in \mathbb{Q}^n \mid C \cdot \vec{p} \geq \vec{h}\}$. A, B, and C are constant integer matrices. $\vec{b}$ and $\vec{h}$ are constant vectors.

Similarly, we have the notion of parametric polytope, which is a bounded parametric polyhedron.

Consider two parameters m and n that are bounded by a polytope

$$\bar{\mathcal{D}}_{(m,n)} = \{(m,n) \in \mathbb{Q}^2 \mid 0 \leq m \leq 100 \wedge 0 \leq n \leq 100\}. \tag{2.4}$$

We can have a parametric polyhedron defined as follows:

$$\mathcal{D}(m,n) = \{(w,i,j) \in \mathbb{Q}^3 \mid w > 0 \wedge 1 \leq i \leq 2m \wedge 1 \leq j \leq n - 2i\}.$$

A parametric polytope can be

$$\bar{\mathcal{D}}_1(m,n) = \{(w,i,j) \in \mathbb{Q}^3 \mid w = 1 \wedge 1 \leq i \leq 2m \wedge 1 \leq j \leq n - 2i\}. \tag{2.5}$$

In this thesis, we are also interested in the number of integer points in a set $\bar{\mathcal{D}}(\vec{p}) \cap \mathbb{Z}^d$, called *cardinality* and denoted by $|\bar{\mathcal{D}}(\vec{p})|$. For a set $\bar{\mathcal{D}} \cap \mathbb{Z}^d$, its cardinality $|\bar{\mathcal{D}}|$ can be obtained as a constant, whereas $|\bar{\mathcal{D}}(\vec{p})|$ is expressed as a piecewise quasi-polynomial. A piecewise quasi-polynomial consists of one or more quasi-polynomials.

Definition 2.1.4 (Quasi-polynomial)**.** A quasi-polynomial $q(x)$ in the integer variables x is a polynomial expression in greatest integer parts of affine expressions in the variables.

Definition 2.1.5 (Piecewise Quasi-polynomial)**.** A piecewise quasi-polynomial $q(\vec{x})$, with $\vec{x} \in \mathbb{Z}^d$ consists of one or more quasi-polynomials. Each quasi-polynomial $q_i(\vec{x})$ is defined only for a disjoint piece $\bar{\mathcal{D}}_i(\vec{x})$ of a parametric polytope $\bar{\mathcal{D}}(\vec{x})$. Each $\bar{\mathcal{D}}_i(\vec{x})$ is also called a *chamber* C_i. For a given point $\vec{x} \in \bar{\mathcal{D}}(\vec{x})$, the piecewise quasi-polynomial evaluates to

$$q(\vec{x}) = \begin{cases} q_i(\vec{x}) & \text{if } x \in \bar{\mathcal{D}}_i(\vec{x}) \\ 0 & \text{otherwise.} \end{cases} \tag{2.6}$$

Consider the parametric polytope $\bar{\mathcal{D}}_1(m,n)$ in Equation (2.5) with parameters m and n bounded by the polytope in Equation (2.4). For the number of integer points in the set $\bar{\mathcal{D}}_1(m,n) \cap\cap \mathbb{Z}^3$, $|\bar{\mathcal{D}}_1(m,n)|$ can be obtained as a piecewise quasi-polynomial as follows:

$$\begin{cases} -2m - 4m^2 + 2mn & \text{if } (m,n) \in C1 \\ -\frac{1}{4}n + \frac{1}{4}n^2 - \frac{1}{2} \cdot \{0,1\}_n & \text{if } (m,n) \in C2 \end{cases}$$

where $\{0,1\}_n$ is called a periodic number with period 2. $C1$ and $C2$ are called chambers given as

$$C1 = \{(m,n) \in \mathbb{Z}^2 \mid 2 + 4m \leq n \wedge 1 \leq m \leq 100 \wedge 0 \leq n \leq 100\},$$
$$C2 = \{(m,n) \in \mathbb{Z}^2 \mid n \leq 1 + 4m \wedge 3 \leq n \leq 100 \wedge 0 \leq m \leq 100\}.$$

Often when we use the polytope mode to represent execution of a program, we need the definition of a lexicographic order.

Definition 2.1.6 (Lexicographic order)**.** Given that two vectors $\vec{a}, \vec{b} \in \mathbb{Z}^n$ are elements of a polyhedron. $\vec{a} \prec \vec{b}$ denotes that $\vec{a}$ is lexicographically smaller than $\vec{b}$, if

$$\bigvee_{i=1}^{n} (a_i < b_i \wedge \bigwedge_{j=1}^{i-1} a_j = b_j)$$

For instance, given $\vec{a} = (w,i,j) = (0,1,3)$ and $\vec{b} = (w,i,j) = (0,2,1)$, we have $\vec{a} \prec \vec{b}$.

When using the polytope model to represent loop nests, we often need to deal with dependence relations to express data dependencies.

Definition 2.1.7 (Dependence Relation [122])**.** A dependence relation R, also called a basic polyhedral map, is defined as

$$R = \{\vec{x}_1 \rightarrow \vec{x}_2 \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \mid \vec{x}_1 \in \mathcal{D}_1 \wedge \vec{x}_2 \in \mathcal{D}_2 \wedge \vec{x}_2 = A\vec{x}_1 + \vec{c}\}, \tag{2.7}$$

where A is an integer matrix and $\vec{c}$ is a constant vector. The polyhedron $\mathcal{D}_1$ is the domain of dependence relation R, denoted by dom$R$. The polyhedron $\mathcal{D}_2$ is the range of dependence relation R, denoted by ran$R$.

For instance, we have a dependence relation

$$\begin{aligned} R = &\{(w1, i1, j1) \rightarrow (w2, i2, j2) \in \mathbb{Z}^3 \times \mathbb{Z}^3 \\ &\mid (w1, i1, j1) \in \mathcal{D}_1 \wedge (w2, i2, j2) \in \mathcal{D}_2 \wedge i2 = i1 - 1 \wedge j2 = j1 + 1 \wedge w1 = w2\}, \end{aligned}$$

where

$$\text{dom}R = \mathcal{D}_1 = \{(w1, i1, j1) \in \mathbb{Z}^3 \mid w1 \geq 0 \wedge 1 \leq i1 \leq 7 \wedge 0 \leq j1 \leq 7 - i1\}$$

and

$$\text{ran}R = \mathcal{D}_2 = \{(w2, i2, j2) \in \mathbb{Z}^3 \mid w2 \geq 0 \wedge 0 \leq i2 \leq 6 \wedge 1 \leq j2 \leq 8 - i2\}.$$

Static Affine Nested Loop Programs (SANLP)

The sequential application specifications considered in this thesis are in the form of *Static Affine Nested Loop Programs* (SANLP).

A SANLP consists of several primitive *functions*. A function is considered as a primitive in this thesis. This means that no explicit parallelization is performed within the function. In general, parallelism within functions can be explored at finer level, e.g., by vectorization [98]. A function serves mainly as the computational part of an application task. Note that there is no restriction on the structure within a function. That means that a function may contain an arbitrary structure of code.

However, restrictions do exist at the level of SANLP, in which functions are called and executed. We summarize the key restrictions of SANLPs as follows.

Definition 2.1.8 (Static Affine Nested Loop Program (SANLP) [41])**.** A static affine nested loop program contains a set of functions, each of which is enclosed by one or more loops and *if*-statements. The loops and *if*-statements have the following restrictions:

- loops have a constant step size;

- loop bounds are affine expressions of the enclosing loop iterators, static parameters, and constants. Static parameters are those whose value cannot change at run-time;
- *if*-statements have affine conditions in terms of the loop iterators, static parameters, and constants;
- index expressions of array references are affine constructs of the enclosing loop iterators, static parameters, and constants;
- the data flow between functions in the loop is explicit, which prohibits that two functions communicate through shared variables invisible at the SANLP level.

An example of a SANLP is shown in Listing 1. Although it is represented using the C syntax, in principle SANLP can be expressed in other forms, such as Matlab [66] or Fortran [101]. Four functions `read_image`, `fiter1`, `filter2`, and `write_image` only exchange data through indexed arrays `img` and `ref_img`. Executing the loop body once is called an *iteration*. For function `read_image`, the polyhedral representation of its execution is given in Equation (2.2) and illustrated in Figure 2.2. The black dots denote individual iterations. According to Definition 2.1.6, iteration $\vec{a} = (w, i, j) = (0, 1, 3)$ is executed before $\vec{b} = (w, i, j) = (0, 2, 1)$, denoted as $\vec{a} \prec \vec{b}$.

PPN

A Polyhedral Process Networks (PPN) [125] is defined as a graph $G = (\mathcal{P}, \mathcal{E})$, where $\mathcal{P}$ is the set of processes and $\mathcal{E}$ is the set of edges. The PPN MoC is a special

```
while(1){
  for (i = 1; i <= 10; i++){         // Width
    for (j = 1; j <= 3; j++){        // Height
      read_image(&img[i][j], &ref_img[i][j]);

      if (j <= 2)
        img[i][j] = filter1(img[i][j]);
      else
        img[i][j] = filter2(img[i][j]);

      write_image(img[i][j], ref_img[i][j]);
} } }
```

Listing 1: An example of a SANLP

case of the Kahn Process Networks (KPN) [64] MoC. That is, PPN processes are synchronized through FIFOs, i.e., any process is blocked when attempting to read from an empty FIFO or write to a full FIFO. In the definition of the KPN MoC, no restriction is imposed on the structure of the KPN processes. In contrast, a PPN process has a particular structure due to the fact that it is automatically derived from a SANLP using the PNgen [125] compiler.

Each function in a SANLP corresponds to a separate process in the derived PPN. If two functions access the same data array through their input/output arguments, they may thus have data dependencies, which is determined by Array Dataflow Analysis (ADA) [41].

The execution of a PPN process is specified using affine nested *for*-loops, called *domain*. Formally, a domain D is defined as a polyhedron following Definition 2.1.1, i.e., $D = \{\vec{I} \in \mathbb{Z}^d \mid A \cdot \vec{I} \geq \vec{b}\}$, where $A \in \mathbb{Z}^{m \times d}$, $\vec{b} \in \mathbb{Z}^d$, $\vec{I}$ is an *iteration vector*, and d indicates the nested-loop depth. At each iteration $\vec{I}$ during the execution of a PPN process P, namely $\vec{I} \in D_P$, P first reads data from input ports (*IP*) in the input port domain D_{IP} if $\vec{I} \in D_{IP}$. Then the process executes the process function (computation) and subsequently writes results to output ports (*OP*) in the output port domain D_{OP} if $\vec{I} \in D_{OP}$. The order of executing different iterations in a process domain is specified by a lexicographic order according to Definition 2.1.6 on page 25. The set of iterations, at which a PPN process writes data to the environment, are called *sink iterations*, denoted by D_{snk}. Furthermore, a dependence relation R_E in Definition 2.1.7 on page 26 is defined for each edge E in a PPN. For an edge E, R_E is specified as $R_E = \{\vec{I} \rightarrow \vec{J} \in \mathbb{Z}^{d1} \times \mathbb{Z}^{d2} \mid \vec{I} \in D_{IP} \wedge \vec{J} \in D_{OP} \wedge \vec{J} = B \cdot \vec{I} + \vec{c}\}$. It indicates that data produced at iteration $\vec{J} \in D_{OP}$ is consumed at iteration $\vec{I} \in D_{IP}$ if output port *OP* is connected to input port *IP* via edge E.

Consider the sequential C program given in Listing 1. The equivalent PPN that can be derived using the PNgen [125] compiler is shown in Figure 2.3. For the behavior of process snk, its process domain is given as

$$D_{snk} = \{(w, i, j) \in \mathbb{Z}^3 \mid w > 0 \wedge 1 \leq i \leq 10 \wedge 1 \leq j \leq 3\}. \tag{2.8}$$

Reading data tokens from input port IP_1 to initialize function argument `in1` of function `write_image` is represented as input port domain

$$D_{IP_1} = \{(w, i, j) \in \mathbb{Z}^3 \mid w > 0 \wedge 1 \leq i \leq 10 \wedge 1 \leq j \leq 2\}.$$

For edge E_5, the dependence relation R_{E_5} is expressed as

$$\begin{aligned} R_{E_5} = &\{(w1, i1, j1) \rightarrow (w2, i2, j2) \in \mathbb{Z}^3 \times \mathbb{Z}^3 \\ &\mid (w1, i1, j1) \in D_{IP_3} \wedge (w2, i2, j2) \in D_{OP_3} \wedge w1 = w2 \wedge i1 = i2 j1 = j2\}, \end{aligned}$$

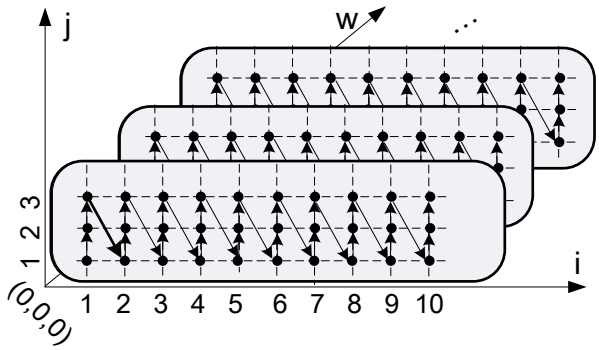


Figure 2.2: The polyhedral representation of the execution of function `read_image` in Listing 1.

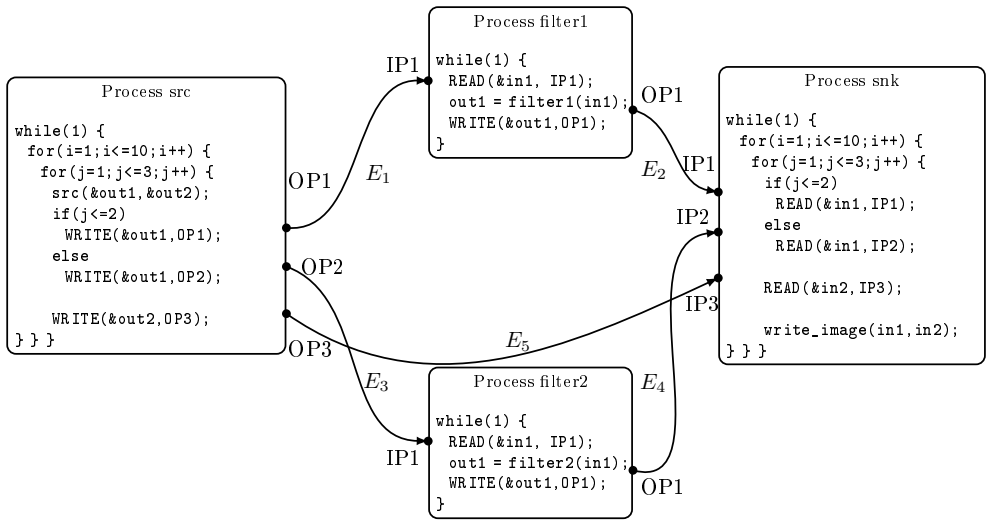


Figure 2.3: PPN corresponding to the SANLP in Listing 1.

where $D_{OP_3} = D_{IP_3} = D_{snk}$.

2.2 Actor-based Data Flow MoCs

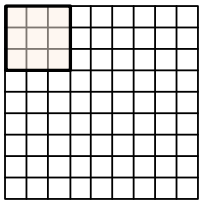
In this section, we give some important definitions concerning the SDF and CSDF MoCs. The related notations are listed in Table 2.3.

2.2.1 Synchronous Data Flow (SDF)

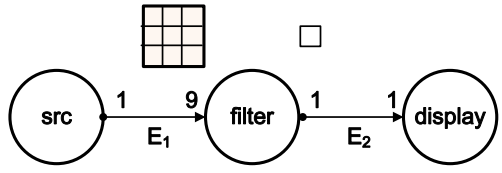
A Synchronous Data Flow (SDF) [75] graph G is defined as $G = (\mathcal{A}, \mathcal{E})$, where $\mathcal{A}$ is the set of actors and $\mathcal{E}$ is the set of edges. For each actor $A_i \in \mathcal{A}$, an execution is called *firing*. It produces/consumes a constant number of data *tokens* to/from edges, denoted by $prd \in \mathbb{N}^+$ and $cns \in \mathbb{N}^+$, respectively. As a special case, the MoC is called Homogeneous Synchronous Data Flow (HSDF) if $prd = cns = 1$ for all production/consumption rates and all actors. To be eligible to fire, each incoming edge E_j of an actor must contain at least cns_j tokens. In this thesis, we assume that auto-concurrent firing of actors are implicitly excluded. We also assume that all cns_j tokens are consumed at the beginning of a firing of an actor. At the end of the firing, all prd_k tokens are produced to each outgoing edge E_k. A token transfered through edges here refers to an atomic data object which can be either an integer or a complex data structure. Tokens are transfered in FIFO fashion. Let us consider for instance the image filter algorithm illustrated in Figure 2.4(a). Its corresponding SDF graph is shown in Figure 2.4(b). At the beginning of the firing, actor filter consumes $3 \times 3 = 9$ pixels from edge E_1 and produces 1 pixel to edge E_2 at the end of the firing.

One important advantage of the SDF MoC is that its functional properties, e.g., *consistency* and *deadlock-free*, can be verified at compile-time. Considering streaming applications which typically execute in a non-terminating fashion, both properties are important to ensure that a given SDF graph can execute indefinitely without causing unbounded token accumulation in FIFOs (buffer overflow), or deadlock. To verify consistency of an SDF graph, a balance equation [75] can be established as follows:

$$\Gamma_G \cdot \vec{q}_G = \vec{0}, \tag{2.9}$$



(a) filter operating on a 3×3 sliding window on the image from left to right and from top to bottom.



(b) An SDF graph G. FIFOs are not illustrated to avoid clutter.

Figure 2.4: An example of an image filter algorithm modeled using the SDF MoC.

A_i	actor
E_i	edge in data flow graph
prd	production rate
cns	consumption rate
PRD	production sequence
CNS	consumption sequence

Table 2.3: Data flow notations.

where Γ_G is called topology matrix and $\vec{q}_G$ is called *repetition vector*. Γ_G is defined as:

$$\Gamma_G = \begin{bmatrix} \Gamma_{1,1} & \cdots & \Gamma_{1,|\mathcal{A}|} \\ \vdots & \Gamma_{j,i} & \vdots \\ \Gamma_{|\mathcal{E}|,1} & \cdots & \Gamma_{|\mathcal{E}|,|\mathcal{A}|} \end{bmatrix} \tag{2.10}$$

with:

$$\Gamma_{j,i} = \begin{cases} prd_j & \text{if actor } A_i \text{ produces to edge } E_j \\ -cns_j & \text{if actor } A_i \text{ consumes from edge } E_j \\ 0 & \text{otherwise.} \end{cases} \tag{2.11}$$

In [75], it is shown that a connected SDF graph is consistent iff $\text{rank}(\Gamma_G) = |\mathcal{A}| - 1$, which ensures Γ_G has a 1-dimensional null space. That is, Equation (2.9) has a non-trivial solution for $\vec{q}_G$. To execute an SDF graph indefinitely with a periodic schedule without unbounded token accumulation, the consistency property is a necessary condition. Consider the SDF graph shown in Figure 2.4(b), its topology matrix Γ_G is given by

$$\Gamma_G = \begin{bmatrix} 1 & -9 & 0 \\ 0 & 1 & -1 \end{bmatrix}$$

Therefore, its repetition vector can be obtained as

$$\begin{aligned} \vec{q}_G &= [q_{src}, q_{filter}, q_{display}] \\ &= [9, 1, 1]. \end{aligned}$$

A consistent SDF graph may still deadlock due to insufficient amount of initial tokens. A SDF graph is said to be deadlocked if none of the actors is eligible to fire at

certain point in time. To detect such a scenario, a periodic admissible schedule [75] can be constructed. If such a schedule does not exist, the SDF graph will deadlock during its execution. Finally, a consistent and deadlock-free SDF graph is said to be *live*. Only live SDF graphs are considered in this thesis.

2.2.2 Cyclo-Static Data Flow (CSDF)

A Cyclo-Static Data Flow (CSDF) [30] graph is similarly defined as $G = (\mathcal{A}, \mathcal{E})$, where $\mathcal{A}$ is the set of actors and $\mathcal{E}$ is the set of edges. CSDF generalizes the SDF MoC by introducing periodically changing token consumption and production rates, called *production/consumption sequence*, denoted by $PRD \in \mathbb{N}^{\phi}$ and $CNS \in \mathbb{N}^{\phi}$, respectively. The production/consumption sequences consist of ϕ *phases*. For the $x$the firing of an actor A_i, it consumes $CNS_j[((x-1) \mod \phi_i)+1]$ tokens from each incoming edge E_j and produces $PRD_k[((x-1) \mod \phi_i)+1]$ tokens to each outgoing edge E_k. PRD_k and CNS_j are defined as $PRD_k = [prd^k_1, \ldots, prd^k_{\phi}]$ and $CNS_j = [cns^j_1, \ldots, cns^j_{\phi}]$, respectively. The length of the production/consumption sequence may vary between CSDF actors. Note that auto-concurrent firing of CSDF actors are implicitly excluded as well.

Similar to the SDF MoC, the consistency of the CSDF MoC is also an important property. For a CSDF graph $G = (\mathcal{A}, \mathcal{E})$, the balance equation [30] is established as follows:

$$\Gamma_G \cdot \vec{r}_G = \vec{0}, \tag{2.12}$$

with

$$\Gamma_{j,i} = \begin{cases} \sum_{k=1}^{k=\phi_i} prd^j_k & \text{if actor } A_i \text{ produces to edge } E_j \\ -\sum_{k=1}^{k=\phi_i} cns^j_k & \text{if actor } A_i \text{ consumes from edge } E_j \\ 0 & \text{otherwise.} \end{cases} \tag{2.13}$$

Assuming $n = |\mathcal{A}|$, the repetition vector $\vec{q}_G = [q_1, \ldots, q_i, \ldots, q_n]$ is then given by

$$\vec{q}_G = Q \cdot \vec{r}_G \text{ with } Q = \mathbb{Z}^{n \times n} \text{ and } Q_{j,i} = \begin{cases} \phi_i & \text{if } j = i \\ 0 & \text{otherwise.} \end{cases} \tag{2.14}$$

ϕ_i is the length of consumption/production sequences of actor A_i. Again, only consistent and deadlock-free, namely live, CSDF graphs are considered in this thesis.

Consider the CSDF graph G_1 in Figure 2.5. The topology matrix of G_1 is given by

$$\Gamma_{G_1} = \begin{bmatrix} 1 & -40 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -66 \end{bmatrix},$$

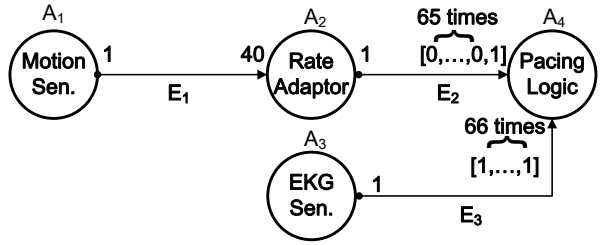


Figure 2.5: A CSDF graph G_1 of a pacemaker application (taken from [99]).

and Q in Equation (2.14) is given by

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 66 \end{bmatrix}.$$

Thus, we can obtain

$$\begin{aligned} \vec{q_G} &= [q_1, q_2, q_3, q_4] \\ &= [40, 1, 66, 66]. \end{aligned}$$

2.3 Hard Real Time Scheduling of Acyclic (C)SDF Graphs

To find a schedule for CSDF graphs where certain performance constraints are guaranteed, periodic schedules are considered to be a common approach. We shall distinguish the periodic schedules considered here from the one defined in [30,75]. The periodic schedule considered in this thesis emphasizes on the fact that the schedule of each actor firing repeats in a strictly periodical way (see Definition 2.3.1). The authors in [22,31] have developed efficient techniques that can derive such Strictly Periodic Schedules (SPS) in polynomial time. Note that the periodic scheduling of the CSDF MoC can be also applied to the SDF MoC, since the CSDF MoC is the superset of the SDF MoC. In particular, the SPS framework for an acyclic CSDF graph developed in [22] is implemented in the Daedalus^{RT} design flow and thus it is considered in this thesis. To ease the discussion of the SPS concept, we use the notations listed in Table 2.4.

Definition 2.3.1 (Strictly Periodic Schedule (SPS))**.** A schedule of a CSDF graph $G = (\mathcal{A}, \mathcal{E})$ is said to be strictly periodic iff

$$\forall A_i \in \mathcal{A} \text{ and } x \in \mathbb{N}^+ \; : \; s_i(x) = S_i + (x-1)T_i, \tag{2.15}$$

where $s_i(x) \in \mathbb{N}^+$ denotes the xth release time of actor A_i, $S_i \in \mathbb{N}$ is the earliest starting time of A_i, and $T_i \in \mathbb{N}^+$ denotes the interval between two consecutive firings of A_i, called *period*.

Essentially, the actors of a CSDF graph under SPS are considered as a set of independent, real-time tasks with implicit deadlines [35]. Therefore, such a real-time task corresponding to a CSDF actor is associated with two parameters, namely period T and earliest starting time S, where the deadline of the task is equal to its period (i.e., implicit deadline).

The main advantage of SPS is that a variety of well-known HRT scheduling algorithms, such as Earliest Deadline First (EDF) [80] or Rate Monotonic (RM) [80], can be applied to temporally schedule CSDF actors allocated on a PE. Meanwhile, temporal isolation of different applications, i.e., different CSDF graphs, that share a single MPSoC platform can be achieved. Moreover, the required platform including the number of PEs and buffer sizes needed to schedule the CSDF graph can be determined in polynomial time.

Under SPS, a firing of a CSDF actor must finish before its deadline which is equal to its period. If the sink actor of a CSDF graph A_{snk} produces *prd* tokens per firing and has a period T_{snk}, the SPS thus guarantees a throughput $\frac{prd}{T_{snk}}$ for the CSDF graph. To compute the period of each actor, the following definition is needed first.

Definition 2.3.2 (Workload of an Actor)**.** The workload of a CSDF actor $A_i \in \mathcal{A}$ per graph iteration, denoted by W_i, is given by $W_i = q_i C_i$, where q_i is the repetition entry of A_i and C_i is the Worst Case Execution Time (WCET) of A_i.

Accordingly, the maximum workload per graph iteration, denoted by $\hat{W}_G$, is defined as $\hat{W}_G = \max_{A_i \in \mathcal{A}}(q_i C_i)$. The minimum period $\check{T}_i$ [22] of an actor A_i under SPS can be computed in linear time as

$$\check{T}_i = \frac{\mathrm{lcm}(\vec{q}_G)}{q_i} \left\lceil \frac{\hat{W}_G}{\mathrm{lcm}(\vec{q}_G)} \right\rceil, \tag{2.16}$$

C_i	Worst-case execution time of an actor A_i
T_i	guaranteed period of an actor A_i
H_i	iteration period of an actor A_i
u_i	utilization of an actor A_i
m	number of PEs

Table 2.4: Notations for HRT scheduling of CSDF MoCs.

where $\text{lcm}(\vec{q}_G)$ is the least common multiple of all repetition entries $q_i \in \vec{q}_G$, and C_i is the WCET of firing a CSDF actor A_i. The minimum period of the sink actor for a CSDF graph determines the maximum throughout that this graph can achieve. To sustain a strictly periodic execution with the period derived by Equation (2.16), the earliest starting time $S_i \in \mathbb{N}$ [22] of an actor A_i can be obtained as

$$S_i = \begin{cases} 0 & \text{if} \quad \text{prec}(A_i) = \emptyset \\ \max_{A_k \in \text{prec}(A_i)}(S_{k \to i}) & \text{otherwise,} \end{cases} \tag{2.17}$$

where $\text{prec}(A_i)$ represents the set of predecessor actors of A_i and $S_{k \to i}$ is given by

$$S_{k \to i} = \min_{t \in [0, S_k + H]} \{ t : \underset{[S_k, \max\{S_k, t\}+d)}{\text{Prd}}(A_k, E_j) \geq \underset{[t, \max\{S_k, t\}+d]}{\text{Cns}}(A_i, E_j),\ \forall d \in [0, H], d \in \mathbb{N} \}, \tag{2.18}$$

where H is defined as an iteration period obtained by $H = q_i T_i$.
$\text{Prd}_{[S_k, \max\{S_k, t\}+d)}(A_k, E_j)$ denotes the total number of tokens produced by actor A_k to edge E_j during the time interval $[S_k, \max\{S_k, t\} + d)$ and
$\text{Cns}_{[t, \max\{S_k, t\}+d]}(A_i, E_j)$ denotes the total number of tokens consumed by actor A_i from edge E_j during the time interval $[t, \max\{S_k, t\} + d]$. In addition, edge E_j connects actors A_k and A_i.

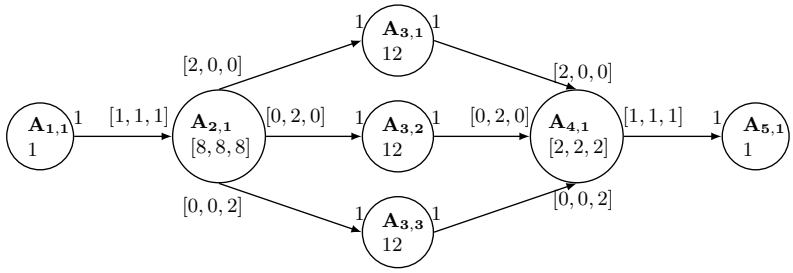
Let us consider the example of the acyclic CSDF graph G_2 in Figure 2.6(a). WCET C_i of each actor is given below the actor name A_i. We can first compute the repetition vector of G_2 in Figure 2.6(a) according to Equation (2.14) on page 32 as:

$$\begin{aligned} \vec{q}_{G_2} &= [q_{1,1}, q_{2,1}, q_{3,1}, q_{3,2}, q_{3,3}, q_{4,1}, q_{5,1}] \\ &= [3, 3, 2, 2, 2, 3, 3] \end{aligned}$$

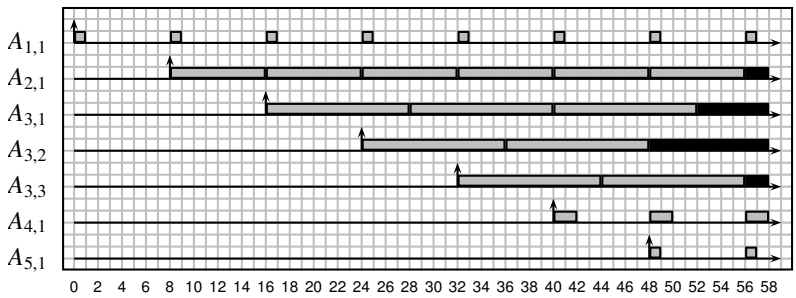
Under SPS, the period of each actor can be obtained using Equation (2.16) as:

$$\begin{aligned} \vec{\check{T}}_{G_2} &= [\check{T}_{1,1}, \check{T}_{2,1}, \check{T}_{3,1}, \check{T}_{3,2}, \check{T}_{3,3}, \check{T}_{4,1}, \check{T}_{5,1}] \\ &= [8, 8, 12, 12, 12, 8, 8] \end{aligned} \tag{2.19}$$

The periodic task-set representation of G_2 is illustrated in Figure 2.6(b). The x-axis represents time. The upper arrows indicate the earliest starting times of individual actors and the grey bars denote WCETs of actor firings. For the sake of discussion, Figure 2.6(b) only illustrates up to time unit 58 on the x-axis and the last firings of actors $A_{2,1}, A_{3,1}, A_{3,2}$ and $A_{3,3}$ are truncated. We can see in Figure 2.6(b) that, after the earliest starting time of each actor, the actor is scheduled in a strictly periodic



(a) A CSDF graph G_2. $A_{1,1}$ and $A_{5,1}$ are considered as the source and sink actors, respectively.



(b) Periodic task-set representation of G_2.

Figure 2.6: An example of a CSDF graph and its real-time task-set representation. Since the execution of the actors repeats indefinitely, the last execution of $A_{2,1}$,$A_{3,1}$,$A_{3,2}$, and $A_{3,3}$ in the figure is truncated and shown in black.

way. For instance, actor $A_{5,1}$ has the earliest starting time $S_{5,1} = 48$. After that, each firing of $A_{5,1}$ occurs every $\check{T}_{5,1} = 8$ time units. Given that $A_{5,1}$ has no outgoing edges and thus it is the sink actor of G_2. Therefore, the maximum throughput of G_2 is $\frac{1}{8}$.

Once periods and earliest starting times of all actors in an acyclic CSDF are derived, the next step is to determine the number of required PEs to schedule the actors and to guarantee that the deadlines (equal to derived periods) of actors are met. To this end, the SPS framework leverages extensively the results from the HRT scheduling theory. Here we only give a brief overview of the HRT scheduling theory that is relevant to this thesis. For the complete treatment of the HRT scheduling topic, please refer to [33]. First, the notion of *utilization* needs to be introduced. Let $G = (\mathcal{A}, \mathcal{E})$ be a CSDF graph, the period of a CSDF actor $A_i \in \mathcal{A}$ be T_i, and WCET of A_i be C_i. The utilization of A_i, denoted by u_i, can be computed as

$$u_i = \frac{C_i}{T_i}, \quad \forall A_i \in \mathcal{A}. \tag{2.20}$$

For instance, using the EDF scheduling algorithm, a set of n actors is schedulable on a PE if the following equation is satisfied [80]:

$$\sum_{i=1}^{n} u_i \leq 1. \tag{2.21}$$

If migration of CSDF actors across PEs is allowed at run-time (global scheduling), the number of required PEs $M(G)$ for a CSDF graph G can be simply computed as

$$M(G) = \left\lceil \sum_{A_i \in \mathcal{A}} u_i \right\rceil. \tag{2.22}$$

In case that no migration of CSDF actors is allowed at run-time (partitioned scheduling), determining the number of required PEs is thus equivalent to the bin-packing problem and can be solved by either *exact* or *approximate* allocation algorithms. An example of an exact allocation algorithm is proposed in [83], which returns an optimal allocation of actors. One disadvantage of using an exact algorithm is its high computational complexity. Therefore, to have a trade-off between optimality of the allocation and computational complexity, an approximate allocation algorithm such as the First-Fit Decreasing (FFD) algorithm [61] can be considered. Let $M_{FFD}(G)$ denote the number of PEs needed for a CSDF graph G under FFD and $M_{OPT}(G)$ denote the number of PEs needed for G using an exact allocation algorithm. It is proven in [134] that the following inequality holds:

$$M_{FFD}(G) \leq \frac{11}{9} M_{OPT}(G) + 1. \tag{2.23}$$

Once allocation of a CSDF graph is determined, the schedule on each PE itself can be built either off-line for efficiency, or on-line for flexibility according to the system requirements.

Let us consider CSDF graph G_2 in Figure 2.6(a). Given $\vec{\tilde{v}}_{T_{G_2}}$ in Equation (2.19), we obtain

$$M(G_2) = \lceil \frac{1}{8} + \frac{8}{8} + \frac{12}{12} + \frac{12}{12} + \frac{12}{12} + \frac{2}{8} + \frac{1}{8} \rceil = 5. \tag{2.24}$$

That is, 5 PEs are required to schedule G_2 using the EDF algorithm and to achieve the maximum throughput of $\frac{1}{8}$.