Chapter 5

SYNCHRONOUS DATA FLOW KERNEL IN SYSTEMC

1. SDF MoC

This chapter describes our implementation of the Synchronous Data Flow (SDF) kernel in SystemC. We present code fragments for the SDF data structure, scheduling algorithms, kernel manipulations and designer guidelines for modeling using the SDF kernel along with an example.

The SDF MoC is a subset of the Data Flow paradigm [32]. This paradigm dictates that a program is divided into blocks and arcs, representing functionality and data paths, respectively. The program is represented as a directed graph connecting the function blocks with data arcs. From [53], Figure 5.1 shows an example of an SDF graph (SDFG). An SDF model imposes further constraints by defining the block to be invoked only when there is sufficient input samples available to carry out the computation by the function block, and blocks with no data input arcs can be scheduled for execution at any time.

In Figure 5.1, the numbers at the head and tail of the arcs represent the production rate of the block and consumption rate of the block respectively, and the numbers in the middle represent an identification number for the arc that we call arc labels. An invoked block consumes a fixed number of data samples on each input data arc and similarly expunges a fixed number of data samples on each of the output data arcs. Input and output data rates of each data arc for a block are known prior to the invocation of the block and behave as infinite FIFO queues. Please note that we interchangeably use function blocks, blocks and nodes for referring to blocks of code as symbolized in Figure 5.1 by A, B, C, D, E, F and G.

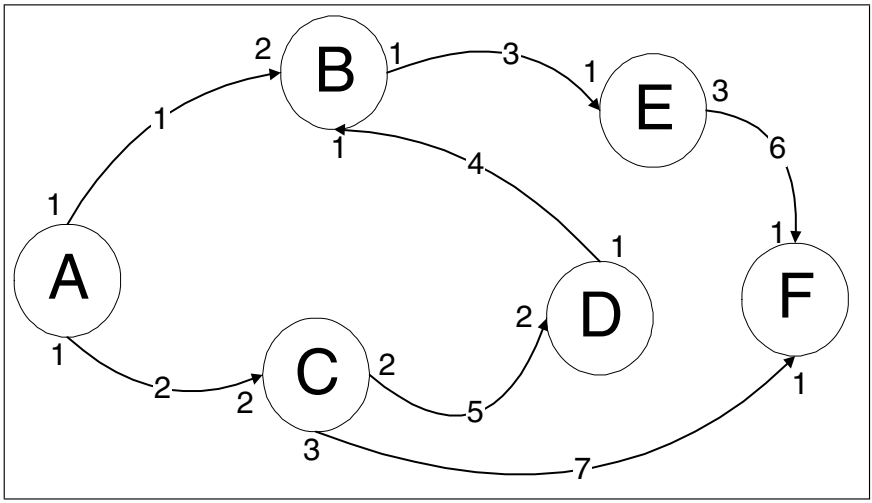


Figure 5.1. Example of a Synchronous Data Flow Graph [53].

Solution to static scheduling and execution of an *executable schedule* for SDF models in SystemC requires solutions to intermediary problems. The problems encountered are as follows:

1 Designing an appropriate data structure to contain information for an SDFG.

2 Constructing *executable schedule*s for the SDFGs. In this problem, there are two sub-problems. They are:

   (a) Computing the number of times each SDF block has to be fired that we refer to as the *repetition vector*.

   (b) Finding the order in which the SDF nodes are be executed, which we term *firing order*.

3 Designing a mechanism for heterogeneous execution of existing Discre te-Event (DE) and SDF kernel.

We define *repetition vector* and *firing order* based on [5]. By *repetition vector* we mean the number of times each function block in the SDFG is to be fired. However, a particular order is needed in which the function blocks in the SDFG are to be fired, that we refer to as the *firing order*. Constructing a *firing order* requires a valid *repetition vector*. A valid *executable schedule* refers to a correctly computed *repetition vector* and *firing order*. The directed nature of the graph and the production and consumption rates provide an algorithm with which the *firing order* is computed.

Problem 2a is discussed in [38] where Lee et al. describe a method whereby static scheduling of these function blocks is computed during compile time rather than runtime. We employ a modification of this technique in our SDF kernel for SystemC explained later in this chapter. The method utilizes the predefined consumption and production rates to construct a set of homogeneous system of linear Diophantine [11] equations. Solution to Problem 2b uses a scheduling algorithm from [5] that computes a *firing order* given that there exists a valid *repetition vector* and the SDFG is consistent. A consistent SDFG is a correctly constructed SDF model whose *executable schedule* can be computed.

We choose certain implementation guidelines to adhere to as closely as possible when implementing the SDF kernel.

1.1 General Implementation Guidelines

Implementation of the SDF kernel in SystemC is an addition to the existing classes of SystemC. Our efforts in isolating the SDF kernel from the existing SystemC kernel definitions introduces copying existing information into the SDF data structure. For example, the process name that is used to distinguish processes is accessible from the SDF data structure as well as existing SystemC process classes. The general guidelines we follow are:

Retain all SystemC version 2.0.1 functionality
: Current functionality that is provided with the stable release of SystemC 2.0.1 should be intact after the alterations related to the introduction of the SDF kernel.

SDF Data structure creation
: All SDF graph structure representation is performed internal to the kernel, hiding the information about the data structure, solver, scheduling algorithms from the designer.

Minimize designer access
: A separate SDF data structure is created to encapsulate the functionalities and behavior of the SDF. The modeler must only access this structure via member functions.

2. SDF Data Structure

Representing the SDF graph (SDFG) needs construction of a data structure to encapsulate information about the graph, such as the production and consumption rates, the manner in which the blocks are connected and so on. In this section, we describe the SDF data structure in detail with figures and code snippets. The majority of our imple-

mentation uses dynamically linked lists (*vector<...>*) provided in the Standard Template Library (STL) [13]. Figure 5.2 shows the SDF data structure. A toplevel list called *sdf_domain* of type *vector<sdf_graph*>* is instantiated that holds the address of every SDF model in the system. This allows multiple SDF models to interact with each other along with the DE models. Furthermore, each *sdf_graph* as shown in Listing 5.1 contains a vector list of pointers to *edges* which is the second *vector* shown in the Figure 5.2. Each *edge* object stores information about an SDF function block whose structure we present later in this section.

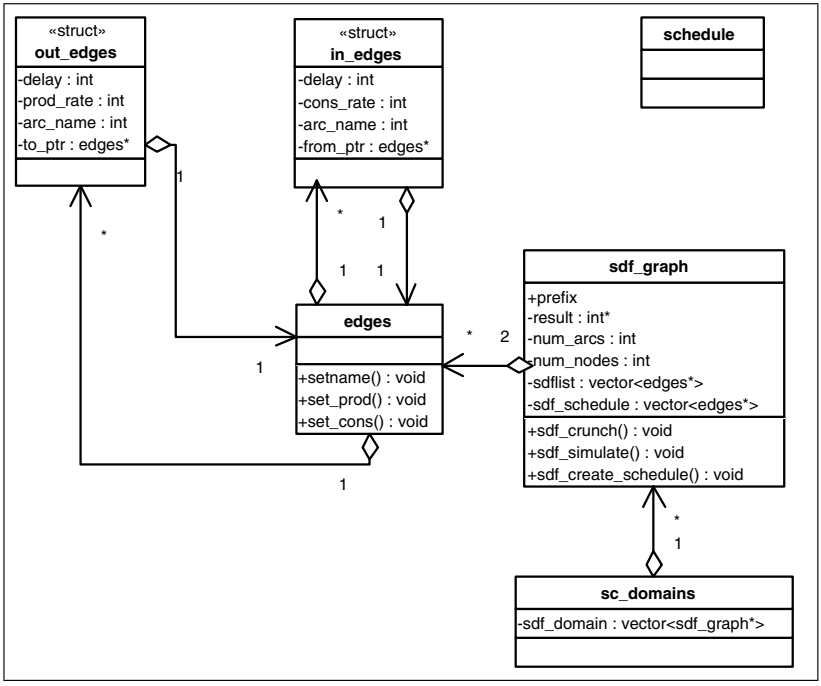


Figure 5.2. SDF Class Diagram

The toplevel class is defined as *sdf_graph* as shown in Listing 5.1. This is the class that holds information pertaining to a single Synchronous Data Flow Graph (SDFG). The SDFG encapsulates the following information: the number of blocks and arcs in the SDF, access to the *executable schedule* via *sdf_schedule*, the *repetition vector* through *result*, a string to identify the toplevel SDF by *prefix*, and a list of the SDF blocks represented by *sdflist* [Listing 5.1, Line 6 - 13].

All SDF blocks are inserted into the *vector<edges*> sdflist* list. This introduces the *edges* class that encapsulates the incoming and the outgoing arcs of that particular SDF block, an integer valued name for con-

Listing 5.1. class sdf_graph

```
class sdf_graph {
  public:
    sdf_graph();     // Constructor & destructor
    ~sdf_graph();

    vector< edges*> sdflist;     // SDF block list
    vector< edges*> sdf_schedule; // executable schedule
    int * result;

    int num_nodes;     // Number of blocks
    int num_arcs;    // Number of arcs

    string prefix;  // Store the name of the SDFG
};
```

structing the *repetition vector* and text based names for comparisons. We typedef this class to *SDFnode*.

Our implementation uses process names for comparison since the SystemC standard requires each object to contain a unique identifying name. When storing the process name either in *sdf_graph* class or *edges* class shown in Listing 5.3, we add a dot character followed by the name of the process.

Listing 5.2. Example showing name()

```
SC_MODULE(test) {

  // port declarations

  void entry1() {name();};
  void entry2() {name();};

  SC_CTOR(test) {
    name();
    SC_METHOD(entry1);  // first entry function
    SC_THREAD(entry2);  // second entry function
    // sensitivity list
  };
};
```

This is necessary to allow multiple process entry functions to be executed when the particular process is found. For clarification, Listing 5.2 presents an example showing SystemC's process naming convention. From Listing 5.2, it can be noticed that there are two entry functions *entry1()* and *entry2()*. Returning the *name()* function from within any of these functions concatenates the process name, and entry function name with a dot character in between. So, calling the *name()* function from *entry1()* will return "*test.entry1*"; calling the same from *entry2()* will return "*test.entry2*" and from the constructor will return "*test*".

Hence, a process name is a unique identifier describing the hierarchy of an entry function, for example "*test.entry1*". We require both these processes to execute for the process name "*test*" and to avoid much string parsing we use the substring matching function *strstr(...)*. However, this will also match a process name other than this module that might have an entry function with a name with "*test*". All processes with a prefix "*test.*" belong to the module "*test*". Therefore, the unique process name is constructed by adding the dot character after the process name and searching for that substring.

Listing 5.3. class edges

```
1 class edges {
2
3   public:
4     vector<out_edges> out;
5     vector<in_edges> in;
6
7     //constructor / destructor
8     edges();
9     ~edges();
10
11    // member functions
12    void set_name(string _name, vector<edges*> & _in); // set
          name
13    void set_prod(sc_method* to_ptr, int _prod); //set
          production rate
14    void set_cons(sc_method* from_ptr, int _cons); //set
          consumption rate
15
16    // variables that will remain public at the moment
17    string name;
18    int mapped_name;
19 };
```

The *edges* class encapsulates the incoming edges to an SDF block and outgoing edges from an SDF block. Lists *vector<out_edges> out* and *vector<in_edges> in* as shown in [Listing 5.3, Line 4 & 5] where *out_edges* and *in_edges* are of C type *struct*s as displayed in Listing 5.4 show the data structure used to store the outgoing arcs and incoming arcs respectively. Every *edge* object stores the process name as a string *name* and a corresponding integer value as *mapped_name* used in creating the topology matrices for the *repetition vector*s.

*struct*s *out_edges* and *in_edges* are synonymous to arcs on an SDFG. The *in_edges* are incoming arcs to a block and *out_edges* are arcs that leave a block [Listing 5.4, Line 1 & 8]. Each arc has an arc label with the integer variable *arc_name*, their respective production and consumption rates and a pointer of type *edges* either to another SDF block or from an SDF block, depending on whether it is an incoming or outgoing arc.

Listing 5.4. struct out_edges & in_edges

```
1 struct out_edges{
2         int prod_rate;  // production rate
3         int arc_name;   // arc label
4         edges* to_ptr;  // pointer to next block
5         int delay;      // delay on this arc
6 };
7
8 struct in_edges {
9         int cons_rate;
10        int arc_name;
11        edges* from_ptr;
12        int delay;
13 };
```

The *struct* and *class* definitions in Listing 5.4 allow us to define an SDF block shown in Figure 5.3.

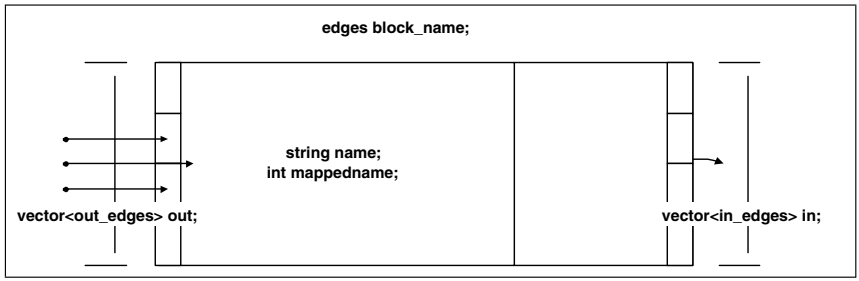


Figure 5.3. Synchronous Data Flow Block.

This representation of an SDF block is instantiated from within an *SC_MODULE()*. This makes an *SC_MODULE()* to be of type SDF method process. This means that one SDF block can only be represented by one *SC_MODULE()*.

We continue to explain the modeling style needed when modeling with the SDF kernel later in this chapter. We also describe the function calls that are required to create an SDF model. We present the prerequisites for the SDF kernel such as the linear Diophantine equation solver, creating a *repetition vector* from the solver and using it to construct a *firing order* yielding an *executable schedule* based on algorithms in [5, 11].

3. Scheduling of SDF

3.1 Repetition vector: Linear Diophantine Equations

The first issue of creating an *executable schedule* is discussed in [38] where Lee et al. describe a method whereby static scheduling of these

function blocks can be computed during compile time rather than run time. The method utilizes the predefined consumption and production rates to construct a set of linear Diophantine [11] homogeneous system of equations and represent it in the form of a topology matrix Γ. It was shown in [38] that in order to have a solution, Γ must be of rank $s-1$ where s is the number of blocks in the SDFG. Solution to this system of equations results in a *repetition vector* for the SDFG. An algorithm used to compute Hilbert's basis [11] solves linear Diophantine equations using the Completion procedure [11, 51] to provide an integer-valued Hilbert's basis. However, the fact that the rank is $s-1$ shows that for SDFs the Hilbert's basis is uni-dimensional and contains only one basis element [38].

Solving linear Diophantine equations is crucial in obtaining a valid *repetition vector* for any SDF graph. A tidy mechanism using the production and consumption rates to construct 2-variable equations and solving this system of equations results in the *repetition vector*. The equations have 2-variables because an arc can only be connected to two blocks. Though this may seem as a simple problem, the simplicity of the problem is challenged with the possibility of the solution of Diophantine equations coming from a real-valued set. This real-valued set of solutions for the Diophantine equations is unacceptable for the purpose of SDF since the number of firings of the blocks require being integral values. Not only do the values have to be integers, but they also have to be non-negative and non-zero, since a strongly connected SDFG can not have a block that is never fired. These systems of equations in mathematics are referred to as linear Diophantine equations and we discuss an algorithmic approach via the Completion procedure with an added heuristic to create the *repetition vector* as presented in [11].

We begin by defining a system of equations parameterized by $\overrightarrow{a} = \{a_i | i = 1...m\}$, $\overrightarrow{b} = \{b_j | j = 1...n\}$ and $\{c, m, n \in \mathbb{N}$ such that the general form for an inhomogeneous linear Diophantine equation is:

$$a_1 x_1 + ... + a_m x_m - b_1 y_1 - ... - b_n y_n = c \tag{5.1}$$

and for a homogeneous Diophantine equation is:

$$a_1 x_1 + ... + a_m x_m - b_1 y_1 - ... - b_n y_n = 0 \tag{5.2}$$

where only integer valued solutions for $\overrightarrow{x}$ and $\overrightarrow{y}$ are allowed. Continuing with the example from Figure 5.1, the arc going from block A to block B via arc label 1 results in Equation 5.3, where u, v, w, x, y and

z represent the number of times blocks A, B, C, D, E, F, and G have to be fired respectively. We refer to the producing block as the block providing the arc with a token and the consuming block as the block accepting the token from the same arc. For arc label 1, the consuming and producing blocks are block B and block A respectively. Therefore, for every arc, the equation is constructed by multiplying the required number of firings of the producing block with the production rate subtracted by the multiplication of the required number of firings of the consuming block with the consumption rate and setting this to be equal to 0.

$$1u - 2v + 0w + 0x + 0y + 0z = 0 \tag{5.3}$$

For the entire SDFG shown in Figure 5.1, the system of equations is described in Equations 5.4. Note that this is a homogeneous system of equations in which the total number of tokens inserted into the system equals the total number of tokens consumed. Our SDF scheduling implementation requires only homogeneous linear Diophantine equations, hence limiting our discussion to only homogeneous Diophantine equations.

$$\begin{aligned}
1u - 2v + 0w + 0x + 0y + 0z &= 0 \\
1u + 0v - 2w + 0x + 0y + 0z &= 0 \\
0u + 1v + 0w + 0x - 1y + 0z &= 0 \\
0u - 1v + 0w + 1x + 0y + 0z &= 0 \\
0u + 0v + 2w - 2x + 0y + 0z &= 0 \\
0u + 0v + 3w + 0x + 0y - 1z &= 0 \\
0u + 0v + 0w + 0x + 3y - 1z &= 0
\end{aligned} \tag{5.4}$$

This system of equations as you notice are only 2-variable equations yielding the topology matrix Γ as:

$$\Gamma = \begin{pmatrix}
1 & -2 & 0 & 0 & 0 & 0 \\
1 & 0 & -2 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & -1 & 0 \\
0 & -1 & 0 & 1 & 0 & 0 \\
0 & 0 & 2 & -2 & 0 & 0 \\
0 & 0 & 3 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 3 & -1
\end{pmatrix}$$

Solving for $\overrightarrow{X}$ in $\Gamma\overrightarrow{X} = 0$ yields the *repetition vector* for the SDFG in Figure 5.1. A linear Diophantine equation solver [51] solves these

topology matrices for system of equations such as in Equations 5.4. The results from the solver for Equations 5.4 are shown in Table 5.1.

Table 5.1. Results from Diophantine Solver

A=u	B=v	C=w	D=x	E=y	F=z
2	1	1	1	1	3

Notice that this methodology of creating a *repetition vector* is specific for acyclic SDF graphs. We discuss SDF graphs with cycles and producing *repetition vector*s for them later in this chapter.

3.2 Linear Diophantine Equation Solver

The problem of solving a system of equations for non-zero, minimal integer-valued solutions was addressed by many mathematicians such as Huet in 1978 [30], Fortenbacher in 1983 [18], Guckenbiehl & Herold in 1985 [26] etc. Of these, A. Fortenbacher and M. Clausen [11] introduce a lexicographic algorithm, which they called the Completion procedure algorithm. We limit our discussion to the Completion procedure algorithm since the labelled digraph approach they discussed is simply an extension of the same concept using labelled digraphs to visualize the problem.

Beginning with some notation, take the general form of a linear inhomogeneous Diophantine equation in Equation 5.1 where *S(a, b, c)* is the set of all nonnegative integer solutions and rewrite it such that the solution set $(\overrightarrow{\xi}, \overrightarrow{\eta}) \in \mathbb{N}^{m+n}$ satisfies the inhomogeneous Diophantine equation $\Sigma_i a_i \xi_i$ - $\Sigma_j b_j \eta_j = c$ for $\{i, j\} \in \mathbb{N}$. Evaluation of the left-hand side of the Diophantine equation is termed as the *defect* at a certain point from the set of $(\overrightarrow{\xi}, \overrightarrow{\eta}) \in \mathbb{N}^{m+n}$ [11]. So, the *defect* of $(\overrightarrow{\xi}, \overrightarrow{\eta})$ is evaluated by $d((\overrightarrow{\xi}, \overrightarrow{\eta})) := \Sigma_i a_i \xi_i$ - $\Sigma_j b_j \eta_j$ where a solution to the equation yields $d((\overrightarrow{\xi}, \overrightarrow{\eta})) = c$. For homogeneous Diophantine equations the equations are similar except that $c = 0$.

The Completion algorithm provided by Fortenbacher and Clausen [11] begins by creating three sets *P, M and Q* representing the set of proposals, the set of minimal solutions and a temporary set, respectively, as shown in Algorithm 5.1. A proposal is the first minimum guess by the algorithm for computing Hilbert's basis. The initialization of P starts with the minimal proposals that are used through the completion procedure. The other two sets M and Q are initially empty sets. The algorithm begins by selecting a proposal P_1 and during the Completion procedure it

increments this proposal according to the *defect* of that proposal. For a proposal $p = (\overrightarrow{\xi}, \overrightarrow{\eta})$, if $d(p) < 0$ then $\overrightarrow{\xi}$ is incremented, and if $d(p) > 0$ then $\overrightarrow{\eta}$ is incremented. If $d(p) = 0$ then a minimal solution is found and this is added to M. A test for minimality is performed and proposals that are not minimal with respect to the computed solution are removed from P. Once there are no more proposals the algorithm terminates. A Pascal implementation was provided in the paper that was converted to a C implementation by Dmitrii Pasechnik [51]. We further converted the C implementation to a C++ implementation

Algorithm 5.1: Completion procedure [11]

{Initialization}
$P_1 := (e_1, \overrightarrow{0}), ..., (e_m, \overrightarrow{0})$
$M_1 := NULL$
$Q_1 := NULL$
{Completion step}
$Q_{k+1} := \{p + (\overrightarrow{0}, e_j) | p \in P_k, d(p) > 0, 1 \leq j \leq n\}$
$\quad \cup \{p + (e_1, \overrightarrow{0}) | p \in P_k, d(p) < 0, 1 \leq i \leq m\}$
$M_{k+1} := \{p \in Q_{k+1} | d(p) = 0\}$
$P_{k+1} := \{p \in Q_{k+1} \setminus M_{k+1} |\ p\ \ minimal\ \ in\ \ p\ \cup\ \bigcup_{i=1}^{k} M_i\}$
{ Termination }
$P_k = NULL?$
$M := \bigcup_{i=1}^{k} M_i$

For our implementation, we employ the same algorithm to solve Diophantine equations with an added heuristic specific for SDFGs. We work through the running example presented in Figure 5.1 to show the added heuristic. Let us first explain why there is a need for a heuristic. Figure 5.1 contains seven equations and six unknowns, over-constraining the system of equations, but Algorithm 5.1 does not explicitly handle more than one equation. Hence, there has to be a way in which all equations can be considered at once on which the Completion procedure is performed. One may speculate an approach where all the equations are added and the *defect* of the sum of all equations is used to perform the algorithm. However, this is incorrect since the *defect* of the sum of the equations can be zero without guaranteeing that the *defect* for the individual equations being zero. This case is demonstrated in Step 3 in Table 5.2. This occurs because the set of solutions when considering the sum of all equations is larger than the set of solutions of the system of equations as described in seven equations, which can be confirmed by taking the rank of the matrices that the corresponding equations yield. Hence, we provide a heuristic that ensures a correct solution for the system of equations.

Algorithm 5.2: Completion procedure with Heuristic

Given m simultaneous 2 variable homogeneous Diophantine equations on a set of n variables, this algorithm finds a solution if one exists.
Let $E = \{(eq_1, eq_2, ..., eq_m)\}$ be the equation set.
Let P be an n tuple of positive integers initially $P := \{\overrightarrow{1}\}$
Let $x_1, x_2, ..., x_n$ be a set of variables.
Let $eq_j \equiv a_j^1 x_{l_j} - a_j^2 x_{k_j} = 0$ where $(j = 1, 2, ..., m)$ and a_j^1 , a_j^2 are coefficients of the j^{th} equation eq_j where $VARS(eq_j) = \{x_l, x_k\}$
Let $rhs(eq_j) = a_j^1 x_{l_j} - a_j^2 x_{k_j}$ which is a function of $\{x_l, x_k\}$
Let $INDICES(eq_j) = \{l_j, k_j\}$
Let d^j be the defect of eq_j evaluated at (α, β) such that $d^j = d(eq_j, \alpha, \beta) = (rhs(eq_j))|_{x_{l_j} \mapsto \alpha, x_{k_j} \mapsto \beta}$ where for any function over two variables u, v, $f(u,v)|_{u \mapsto a, v \mapsto b} = f(a,b)$
$D = \{d(eq_j, p_{l_j}, p_{k_j}) \mid VARS(eq_j) = \{l_j, k_j\}\}$ {So D is an m tuple of m integers called the defect vector}
$MAXINDEX(D) = j$ where $j = min\{r \mid d(eq_r, p_{l_r}, p_{k_r}) \geq d(eq_n, p_{l_n}, p_{k_n}) \, \forall \, n \in \{1, 2, ..., m\}$ *and* $\{l_n, k_n\} = INDICES(eq_n)\}$
{Repeat until defect vector D is all zeros or the maximum proposal in P is less then or equal to the lowest-common multiple of all the coefficients in all equations}
while $((D! = \overrightarrow{0}) \bigwedge (max(\overrightarrow{p}) \leq (lcm_{(r=1,2),(j=1..m)}(a_j^r)))$ **do**
  $j = MAXINDEX(D)$
  if $(d^j < 0)$ **then**
    Let x be the l^{th} variable and
    $(x, y) = VARS(eq_j)$
    for *all* $eq_i \in E$ **do**
      if $(x \in \{VARS(eq_i)\})$ **then**
        Re-evaluate d^i with $[p_l \mapsto p_l + 1]$
      end if
    end for
    Update $p_l \in P$ with $[p_l \mapsto p_l + 1]$
  else
    if $(d^j > 0)$ **then**
      Let y be the k^{th} variable and
      $(x, y) = VARS(eq_j)$
      for *all* $eq_i \in E$ **do**
        if $(y \in \{VARS(eq_i)\})$ **then**
          Re-evaluate d^i with $[p_k \mapsto p_k + 1]$
        end if
      end for
      Update $p_k \in P$ with $[p_k \mapsto p_k + 1]$
    end if
  end if
end while

The heuristic we implement ensures that before processing the *defect* of the proposal, the proposal is a vector of ones. By doing this, we indicate that every function block in the SDFG has to fire at least once.

After having done that, we begin the Completion procedure by calculating the *defect* of each individual equation in the system of equations and recording these values in a *defect* vector D. Taking the maximum *defect* from D defined as d^j for that j^{th} equation, if $d^j > 0$ then the correct variable for j is incremented and if $d^j < 0$, then the appropriate variable for the j is incremented for the j^{th} equation. However, updating the *defect* d^j for that equation is not sufficient because there might be other occurrences of that variable in other equations whose values also require being updated. Therefore, we update all occurrences of the variable in all equations and recompute the $defect$ for each equation. We perform this by checking if the updated variable exists in the set of variables for every equation extracted by the *VARS()* sub-procedure and if that is true, that equation is re-evaluated. The algorithm repeats until the *defect* vector $D = \overrightarrow{0}$ terminating the algorithm. The algorithm also terminates when the lowest-common multiple of all the coefficients is reached without making the $defect$ vector $D = \overrightarrow{0}$ [11]. This is because none of the proposal values can be larger than the lowest-common multiple of all the coefficients[11]. For Figure 5.1, the linear Diophantine equations are as follows: $u - 2v = 0, u - 2w = 0, v - y = 0, v + x = 0, 2w - 2x = 0, 3w - z = 0, 3y - z = 0, -u - z = 0$ and the *repetition vector* from these equations is shown in Table 5.1

To further clarify how the algorithm functions, we walk through the example in Figure 5.1 and compute the *repetition vector*. We define the proposal vector P as $\overrightarrow{p} = (u, v, w, x, y, z)$ where the elements of $\overrightarrow{p}$ represent the number of firings for that particular block. Similarly, we define a $defect$ vector $\overrightarrow{d} = (eq_1, eq_2, eq_3, eq_4, eq_5, eq_6, eq_7)$ where eq_n is the n^{th} equation in the system of equations. The system of equations are restated below:

$$
\begin{aligned}
eq_1 &= 1u - 2v + 0w + 0x + 0y + 0z = 0 \\
eq_2 &= 1u + 0v - 2w + 0x + 0y + 0z = 0 \\
eq_3 &= 0u + 1v + 0w + 0x - 1y + 0z = 0 \\
eq_4 &= 0u - 1v + 0w + 1x + 0y + 0z = 0 \\
eq_5 &= 0u + 0v + 2w - 2x + 0y + 0z = 0 \\
eq_6 &= 0u + 0v + 3w + 0x + 0y - 1z = 0 \\
eq_7 &= 0u + 0v + 0w + 0x + 3y - 1z = 0
\end{aligned}
$$

and steps in processing this system of equations with Algorithm 5.2 are shown in Table 5.2.

In Table 5.2, the proposal vector begins as a vector of all ones with its computed *defect*. The next equations to consider are the ones with

Table 5.2. Solution steps for example using Completion procedure

Step	$\overrightarrow{p}$	$\overrightarrow{d}$
1	(1, 1, 1, 1, 1, 1, 1)	(-1, -1, 0, 0, 0, 2, 2)
2	(1, 1, 1, 1, 1, 1, 2)	(-1, -1, 0, 0, 0, 1, 1)
3	(1, 1, 1, 1, 1, 1, 3)	(-1, -1, 0, 0, 0, 0, 0)
4	(2, 1, 1, 1, 1, 1, 3)	(0, 0, 0, 0, 0, 0, 0, 0)

the highest individual defect. We perform the Completion procedure on equations with the highest *defect*, that are eq_6 & eq_7, and increment z twice, to reduce the defect of eq_6 & eq_7 to zero. Then, we reduce the negative $defect$s for eq_1 & eq_2, that we compensate by incrementing v. This results in $defect$ vector being all zeros completing the completion procedure and giving a *repetition vector*. Though this example is strictly for acyclic SDFGs, we use the same algorithm in solving cyclic SDFGs. We present our discussion and algorithms for creating *executable schedule*s for cyclic SDF graphs in the next section.

3.3 Firing Order: repetition vectors for non-trivial cyclic SDF graphs

Most DSP systems have feedback loops and since the SDF MoC is used for DSP, we expect occurrences of loops in SDFGs. These feedback loops are represented as cycles in an SDFG and it is necessary for our SDF kernel to be able to efficiently handle these types of cycles. In this section we explain how cycles affect the *repetition vector* and discuss the *firing order* produced by the algorithm developed in [5].

We previously described the algorithm used in creating a *repetition vector* and we employ the same algorithm in calculating the *repetition vector* for cyclical SDFGs. However, the order in which these blocks have to be fired needs to be computed. The *firing order* is important because the SDF paradigm requires a specific order in which the function blocks are executed. SystemC's DE kernel schedules its processes in an unspecified manner, so it could schedule block F from Figure 5.4 as the first block for execution, which is incorrect for the SDF model. The *repetition vector* only describes the number of times F needs to be fired and not in which order F is fired. Until the proper *firing order* is found the system would deadlock due to F not having enough input on the incoming arcs from blocks C and E. We can see that blocks C and E have to be fired before F can be fired to correctly execute the SDFG.

For acyclic graphs as shown in Figure 5.4, one can use a topological sorting via Depth-First Search (DFS) algorithm to compute the *firing order*. However, in the presence of a cycle, a topological ordering does

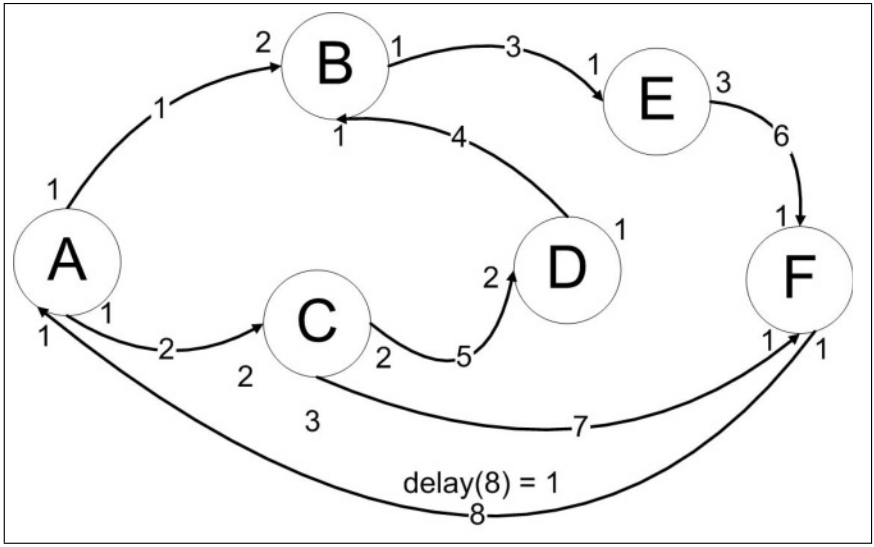


Figure 5.4. Example of a cyclic Synchronous Data Flow Graph [53].

not exist. Since, our goal is to obtain the *firing order* regardless of the SDFG containing cycles or not we employ another algorithm in constructing the *firing order*.

Bhattacharyya, Murthy and Lee in [5] developed scheduling algorithms for SDF of which one scheduling algorithm determines the *firing order* of non-trivial (cyclic or acyclic) SDFGs. However, before we present scheduling Algorithm 5.3, we discuss the *delay* terminology for an SDFG shown in Figure 5.4. Note the production rate at the head of the arcs, consumption rate at the tail of the arcs (marked by the arrow head), arc label or name in the middle of the arc and a newly introduced $delay(\alpha) = \gamma$ where α is an arc label and γ is the *delay* on that arc. *delay* [5] represents the number of tokens the designer has to inject initially into arc α for the SDF model to execute. The concept of *delay* is necessary when considering cyclical SDFGs. This is because of the additional constraint set by the SDF paradigm that every block only executes when it has sufficient input tokens on its input arcs, thus a cycle as indicated by arc label 8 in Figure 5.1 without the *delay* causes the SDFG to deadlock. Hence, no blocks can be fired. The *delay* acts as an initial token on that arc to allow simulation to begin. We introduce the concept of *delay* on all arcs, except we omit displaying the *delay*s that are zero, like on arc label 1 and 2 and so on.

Scheduling Algorithm 5.3 creates a *firing order* by using the *repetition vector* from the Hilbert's solver. If the *firing order* is valid then the SDF

kernel executes the SDF processes in the correct sequence for the correct number of times. The simulation terminates if either the *repetition vector* is invalid or the SDFG is inconsistent. An SDFG is inconsistent when the number of times every block scheduled in the *firing order* is not reflected by the number of times the block is supposed to be scheduled for execution as per the *repetition vector*.

Algorithm 5.3: Construct Valid Schedule Algorithm [5]
Require:
Let *ready* to be a queue of function block names
Let *queued* and *scheduled* to be vectors of non-negative integers indexed by the function blocks in SDFG
Let S be a schedule and initialize S to null schedule
Let **state** be a vector representing the number of tokens on each edge indexed by the edges
Let **rep** be a vector showing results from the Diophantine equation solver computed by Algorithm 5.2 (*repetition vector* indexed by the function block names)

for *all function blocks in SDFG* **do**
  for *each incoming edge* α **do**
    store delay on α to $state(\alpha)$
  end for

  for *each outgoing edge* β **do**
    store delay on β to $state(\beta)$
  end for
end for
for *each function block N in SDFG* **do**
  save **rep** for N in temporary variable *temp_rep*
  for *each incoming edge* α **do**
    set *del_cons* equal to $\lfloor delay(\alpha)/cons\_rate(\alpha) \rfloor$
    $temp\_rep = min(temp\_rep, del\_cons)$
  end for
  if $temp\_rep > 0$ **then**
    store *temp_rep* in $queued(N)$
    store 0 in $scheduled(N)$
  end if
end for

while *ready* is not empty **do**
  pop function block N from *ready*
  add $queued(N)$ invocations to S
  increment $scheduled(N)$ by value of $queued(N)$
  store *temp_n* to $queued(N)$
  set $queued(N)$ value to 0
  for *each incoming edge* α *of function block* N **do**
    set **state** for α is decremented by ($temp\_n \times cons\_rate$ on (α))
  end for

```
    for each outgoing edge β of function block N do
      set state for β is incremented by (n × prod_rate on (β))
    end for
    for each ougoing edge α of function block N do
      to_node is the function block pointed by α
      temp_r = subtract rep for to_node by scheduled(to_node)
    end for
    for each incoming edge γ of to_node(α) do
      set del_cons to ⌊state value for γ / cons_rate on γ⌋
    end for
    if (temp_r > queued(to_node)) then
      push to_node to ready
      set queued(to_node) to temp_r
    end if
  end while
  for each function block N in SDFG do
    if (scheduled vector for N ≠ to rep for N) then
      Error::Inconsistent Graph
      Exit
    end if
  end for
  S contains schedule
```

In Algorithm 5.3, ***state*** is a vector that initially contains the *delay*s on each of the arcs in the SDFG and is indexed by the arc names labelled α or β. During execution, ***state*** denotes the number of tokens on each arc. Similarly, *queued* and *scheduled* are vectors indexed by function block name N for the purpose of storing the number of times the block is *scheduled* and the number of times a block has to be scheduled to be fired is *queued*, respectively. *rep* is a temporary pointer to point to the *repetition vector* and *ready* holds the blocks that can be fired. The algorithm iteratively determines whether the SDFG is consistent and if so, results in an *executable schedule*.

The initialization begins by first traversing through all the function blocks in the SDFG and setting up the ***state*** vector with its corresponding *delay* for all the arcs on every block. Once this is done, every block is again traversed and for every incoming arc, the minimum between the *repetition vector* for that block and the $delay(\alpha)/cons\_rate(\alpha)$ is sought. If this minimum is larger than zero, then this block needs to be scheduled and added to *ready* since that means it has sufficient tokens on the incoming arcs to fire at least once. This initialization also distinguishes the blocks that are connected in cycles with sufficient *delay* values and schedules them. Scheduling of the remaining blocks is performed in a similar fashion with slight variation.

If the SDFG is consistent, the first block from *ready* is popped and appended to the *schedule* for the number of times the block is to be invoked. For all the incoming edges of this block the *delay* for this arc is recalculated. For the outgoing edges a similar calculation is done except this time the ***state*** for the arc is incremented by the production rate multiplied by *temp_n*. Basically, the algorithm looks at all the outgoing edges and the blocks pointed by these outgoing edges and proceeds to traverse focusing on all blocks pointed by the outgoing edges of the block just popped of the *ready*. Finally, a check for inconsistency is performed where the number of times each block is scheduled has to be equivalent to the number of times it is supposed to be fired from the *repetition vector*. The algorithm concludes with the correct schedule in S for a consistent SDFG.

This scheduling algorithm yields the schedule in the correct *firing order* with the number of times it is to be fired. The kernel will fire according to this schedule. Acyclic and cyclic SDFGs are handled correctly by this algorithm. The final *executable schedule* is stored in *sdf_schedule* accessible to the kernel for execution.

4. SDF Modeling Guidelines

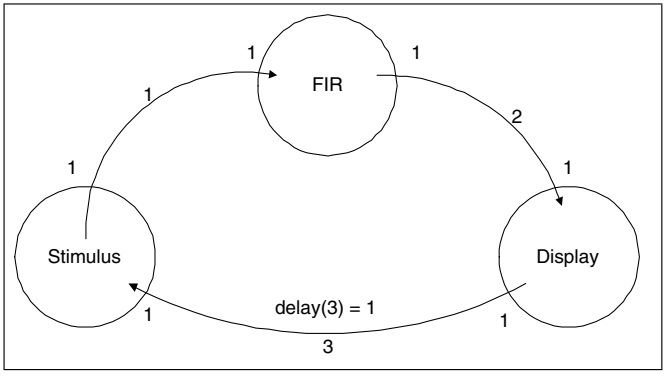


Figure 5.5. FIR Example of a Synchronous Data Flow Graph [53].

Efforts in reducing difficulty of modeling other MoCs have been a key consideration in implementing the SDF kernel. Though, we believe that we have reduced the level of difficulty in designing SDF models in SystemC by increasing the modeling fidelity, we also provide guidelines in creating SDF models. A simple example with full source code segments demonstrates the style. We use the FIR example from Figure 5.5 to compare the FIR example provided with the SystemC 2.0.1 distribution modeled with the DE kernel (that has already been shown in Chapter

3) with the converted model using the SDF kernel. We edit the source code to remove some *std :: cout* statements to make the output from DE FIR example to match the output of the SDF FIR model, but the functionality of the FIR in DE and SDF remains the same. *fir_const.h* is not included in the listings since it is a list of constants that will be available in the full source prints at [36].

Listing 5.5. stimulus.h

```
1 #include <queue>
2 extern sdf_graph sdf1;
3 using namespace std;
4
5 SC_MODULE(stimulus) {
6   edges stimulus_edge;
7   sc_int<8>      send_value1;
8   SC_CTOR(stimulus)  {
9     stimulus_edge.set_name(name(), sdf1.sdflist);
10    SC_METHOD(entry);
11      send_value1 = 0;
12   }
13  void entry();
14 };
```

Notice in Listing 5.5 that the Stimulus block has no input or output ports along or control signal ports. This refers to declarations of ports using *sc_in<...>* or *sc_out<...>*. These are no longer required in an SDF model because static scheduling does not require control signals and our method of data passing is through STL *queue<...>* structures. We do not strictly enforce their removal and if the designer pleases to use SystemC channels and ports for data paths then that can also be employed. However, the SDF kernel statically schedules the SDF blocks for execution at compile time, hence there is no need for one block to signal to the following block when data is ready to be passed on, obviating the need for control signals. For data paths, using signals and channels in SystemC generate events reducing simulation efficiency. Our advised approach is to use *queue<...>* STL queues to transfer data within the SDF model instead of using SystemC. Only the data that has to be passed from one block to another requires an instantiation of a queue such as the *extern queue<int> stimulusQ* [Listing 5.6, Line 6]. Instantiation of *stimulus_edge* object is crucial in defining this *SC_MODULE()* as an SDF method process [Listing 5.5, Line 5 & 6]. This object is used to pass the name of the *SC_MODULE()* and the SDFG which this block belongs to as shown in [Listing 5.5, Line 9].

The queues used in the FIR model are *stimulusQ* and *firQ*, where *stimulusQ* connects the Stimulus block to the FIR block and *firQ* connects from the FIR block to the Display [Listing 5.8, Line 4 & 5]. However,

Listing 5.6. stimulus.cpp

```
#include <systemc.h>
#include "stimulus.h"
#include <queue>

using namespace std;
extern queue<int> stimulusQ;

void stimulus::entry() {
  stimulusQ.push(send_value1);
  send_value1++;
}
```

instantiation of these queues has to be done in a particular manner. Since every arc connects two blocks implies that two blocks must have the same queue visible to them. So, *stimulusQ* should be accessible by the Stimulus block as well as the FIR block. The easiest approach is to instantiate these queues globally in the toplevel file and make them *extern* keyword when a block needs to refer to the *queue<...>*s (if in a different file). With this method all the files must be a part of one compilation since *extern* informs the compiler that the particular variable (in this case *queue<...>*) is instantiated in some other file external to the current scope. Furthermore, memory is not allocated when *extern* keyword is encountered because the compiler assumes that the variables with *extern* keyword have been properly defined elsewhere.

Listing 5.7. fir.h

```
#include <queue>
extern sdf_graph sdf1;
using namespace std;

SC_MODULE(fir) {
  sc_int<9> coefs[16];
  sc_int<8> sample_tmp;
  sc_int<17> pro;
  sc_int<19> acc;
  sc_int<8> shift[16];
  edges fir_edge;
  SC_CTOR(fir)  {
      fir_edge.set_name(name() /*"process_body"*/, sdf1.sdflist)
          ;
      SC_METHOD(entry);
      #include "fir_const.h"
      for (int i = 0; i < 15; i++)  {
            shift[i] = 0;
      }
  void entry()
  };
```

Every SDF block (SDF *SC_MODULE()*) must also have access to the SDFG that it is to be inserted in. This means the *sdf_graph* object that is instantiated globally must be available to the *SC_MODULE()*s such as in [Listing 5.7, Line 2]. However, the integral part of the *SC_MODULE()* declaration is the instantiation of the *edges* object as shown in [Listing 5.7, Line 11]. This object is accessed by the SDF kernel in determining certain characteristics during scheduling. These characteristics are set by member functions available in the *edges* class. One of the first member functions encountered in the *SC_MODULE()* declaration is the *set_name(...)* [Listing 5.7, Line 13] function that is responsible for providing the *edges* object with the module name and the storage list of SDF method processes. The *name()* function from within the *SC_MODULE()* returns the name of the current module. In [Listing 5.7, Line 13] *sdf1.sdflist* is the list where the addresses of the *SC_METHOD()* processes are stored for access by the SDF kernel. Apart from those alterations, the structure of an *SC_MODULE()* is similar to regular SystemC processes.

Listing 5.8. fir.cpp

```
1#include <systemc.h>
2#include "fir.h"
3
4extern queue<int> stimulusQ;
5extern queue<int> firQ;
6
7void fir::entry() {
8   sample_tmp = stimulusQ.front();   stimulusQ.pop();
9   acc = sample_tmp*coefs[0];
10  for(int i=14; i>=0; i--) {
11     pro = shift[i]*coefs[i+1];
12     acc += pro;
13  }
14  for(int i=14; i>=0; i--) {
15     shift[i+1] = shift[i];
16  }
17  shift[0] = sample_tmp;
18  firQ.push((int) acc);
19}
```

Note that the functions used to insert data onto the *queue*<...>s are STL functions *push()* and *pop()* [Listing 5.8, Line 8 & 18]. There is also no check for the number of tokens ready to be received by each block. Naturally, this is not required since we are statically scheduling the SDF blocks for an appropriate number of times according to their consumption and production rates. However, this burdens the designer with the responsibility of carefully inserting sufficient tokens on the *queue*<...>s to ensure the simulation does not attempt at using invalid data.

Listing 5.9. display.h

```
1#include <queue>
2extern sdf_graph sdf1;
3using namespace std;
4
5SC_MODULE(display) {
6   int i, tmp1;
7   edges display_edge;
8   SC_CTOR(display)
9     { display_edge.set_name(name() /*"display"*/, sdf1.sdflist);
10      SC_METHOD(entry);
11      i = 0;
12    }
13  void entry();
14};
```

Listing 5.10. display.cpp

```
1#include <systemc.h>
2#include "display.h"
3extern queue<int> firQ;
4void display::entry(){
5   tmp1 = firQ.front();   firQ.pop();
6   cout << tmp1 << endl;
7   i++;
8   if(i == 5000000) {
9     sc_stop();
10  };
11}
12// EOF
```

The SDF kernel requires the modeler to specify the terminating value as in the DE kernel example. This is similar to the termination situations posed in [23]. However, we define a period of SDF as a complete execution of the SDF. In this example since there is a cycle, every period is an execution of the SDF model. We halt the execution after a specified number of samples using the *sc_stop()* [Listing 5.10, Line 9] which tells the kernel that the modeler has requested termination of the simulation.

The toplevel *SC_METHOD()* process labelled *toplevel* in [Listing 5.12, Line 2] constructs the SDF graph encapsulating that entire SDF model inside it. The choice of the toplevel process can be of any SystemC type. The entry function has to be manipulated according to the process type since they continue to follow SystemC semantics. The construction of this module is straightforward whereby pointers to each of the SDF methods are member variables and are initialized to their corresponding objects in the constructor. The important step is in constructing the SDFG within the constructor (*SC_CTOR()*) since the constructor is only invoked once per instantiation of the object. The functions *set_prod(...)* and *set_cons(...)* set the arcs on the SDFG. Every *SC_MODULE()* de-

Listing 5.11. main.cpp

```
1#include <systemc.h>
2#include "stimulus.h"
3#include "display.h"
4#include "fir.h"
5sdf_graph sdf1;
6queue <int> stimulusQ;
7queue <int> firQ;
8// General METHOD process to verify proper execution of ordinary
      DE METHODs
9SC_MODULE(foo) {
10   sc_in_clk clock;
11   void message()  {
12     cout << sc_time_stamp() << " foo with next_trigger executed
           " << endl;
13     next_trigger(2, SC_NS);
14   }
15     SC_CTOR(foo){
16       SC_METHOD(message)  {
17         sensitive << clock.pos();
18         };
19   }
20};
21// General CTHREAD process to verify proper execution of
      ordinary DE THREADs
22SC_MODULE(foo_cthread)  {
23   sc_out<bool> data_sdf;
24   sc_in_clk clock;
25   void msg () {
26     bool b= false;
27     while(1)  {
28       cout << sc_time_stamp() << " CTHREAD executed " << endl;
29       wait(3);
30       cout << "Instruct SDF to fire" << endl;
31       if (b == true)
32         b = false;
33       else
34         b = true;
35       data_sdf.write(b);
36     }
37   }
38   SC_CTOR(foo_cthread) {
39     SC_CTHREAD(msg, clock.pos()) {
40       sensitive << clock.pos() ;
41       };
42     }
43};
```

fines an SDF block that requires the arcs being set. The arguments of these set functions are: the address of the *edges* object instantiated in a module that it is pointed to or from, the production or consumption rate depending on which function is called and the *delay*. We also enforce a global instantiation of *sdf_graph* [Listing 5.11, Line 5] type object for every SDFG that is present in the model. Using the *schedule* class and the *add_sdf_graph(...)*, the SDFG is added into a list that is visible by the overlaying SDF kernel [Listing 5.12, Line 23]. In addition, this toplevel process must have an entry function that calls *sdf_trigger()* [Listing 5.12,

Listing 5.12. toplevel and main()

```
1// Top Level METHOD encapsulating the SDFG
2SC_MODULE(toplevel) {
3 sc_in<bool> data;
4  fir* fir1; //( "process_body*");
5  display* display1 ; //( "display");
6  stimulus* stimulus1; //("stimulus_block");
7  void entry_sdf()  {
8    sdf_trigger(name());
9  }
10  SC_CTOR(toplevel) {
11    SC_METHOD(entry_sdf)
12      sensitive << data ;
13
14    fir1 = new fir("process_body*");
15    display1 = new display("display");
16    stimulus1 = new stimulus("stimulus");
17    stimulus1->stimulus_edge.set_prod(&fir1->fir_edge,1 , 0);
18    fir1->fir_edge.set_cons(&stimulus1->stimulus_edge, 1, 0);
19    fir1->fir_edge.set_prod(&display1->display_edge, 1, 0);
20    display1->display_edge.set_cons(&fir1->fir_edge, 1, 0);
21    display1->display_edge.set_prod(&stimulus1->stimulus_edge
          , 1, 1);
22    stimulus1->stimulus_edge.set_cons(&display1->display_edge
          , 1, 1);
23    schedule::add_sdf_graph(name(), &sdf1);
24  }
25};
26
27int sc_main (int argc , char *argv[]) {
28  sc_clock          clock;
29  sc_signal<bool> data;
30  toplevel top_level("top_level*sdf*");
31  top_level.data(data);
32  foo foobar("foobar");
33  foobar.clock(clock);
34  foo_cthread foo_c("foo_C");
35  foo_c.clock(clock);
36  foo_c.data_sdf(data);
37  sc_start(-1);
38 return 0;
39}
```

Line 8] signalling the kernel to process all the SDF methods corresponding to this toplevel SDF module. These guidelines enable the modeler to allow for heterogeneity in the models since the toplevel process can be sensitive to any signal that is to fire the SDF. The *SC_CTHREAD()* [Listing 5.11, Line 39] partakes in this particular role where every three cycles the SDF model is triggered through the signal *data*. However, the designer has to be careful during multi-MoC modeling due to the transfer of data from the DE blocks to the SDF blocks. This is because there is no functionality in the SDF kernel or for that matter even the DE kernel that verifies that data on an STL *queue*<...> path is available before triggering the SDF method process. This has to be carefully handled by the designer. These style guides for SDF are natural to the paradigm

and we believe that this brief explanation of modeling in SDF provides sufficient exposure in using this SDF kernel along with the existing DE kernel.

4.1 Summary of Designer Guidelines for SDF implementation

The designer must remember the following when constructing an SDF model:

- To represent each *SC_MODULE()* as a single SDF function block as in Listing 5.5.
- Ensure that each process type is of *SC_METHOD()* process [Listing 5.5, Line 10].
- The module must have access to the instance of *sdf_graph* that it is to be inserted in [Listing 5.7, Line 2].
- An object of *edges* is instantiated as a member of the *SC_MODULE()* and the *set_name()* function is called with the correct arguments [Listing 5.7, Line 11 & 13].
- The instance of *sdf_graph* is added into the SDFG kernel list by calling the *static add_sdf_graph()* function from *schedule* class as in [Listing 5.12, Line 23].
- Set *delay* values appropriately for the input and output samples on the *queue*<...> channels for every arc [Listing 5.12, Line 22].
- Introduce a *sc_clock* object to support the timed DE MoC [Listing 5.12, Line 28].
- Ensure that the entire SDFG is encapsulated in a toplevel process of any type [Listing 5.12, Line 2 - 39].
- Toplevel process must have an entry function that calls *sdf_trigger()* ensuring that when this process is fired, its corresponding SDF processes are also executed [Listing 5.12, Line 8].

5. SDF Kernel in SystemC

We implement the SDF kernel and update the DE kernel function calls through the use of our API discussed in Chapter 8. We limit our alterations to the original source, but some change in original source code is unavoidable. Our approach for kernel implementation is in a particular manner where the SDF kernel exists within the Discrete-Event kernel.

Listing 5.13. split_processes() function from API class

```
void sc_domains::split_processes() {
  sc_process_table* m_process_table = de_kernel->
      get_process_table();
  const sc_method_vec& method_vec = m_process_table->method_vec
      ();
  if (sdf_domain.size() != 0) {
    for (int sdf_graphs = 0; sdf_graphs < (signed) sdf_domain.
        size() ; sdf_graphs++) {
      // Extract the address of SDFG
      sdf_graph * process_sdf_graph = sdf_domain[sdf_graphs];
      sdf* process_sdf = &process_sdf_graph->sdflist;
      for( int i = 0; i < method_vec.size() ; i++ ) {
        sc_method_handle p_method_h;
        sc_method_handle method_h = method_vec[i];
        if( method_h->do_initialize() ) {
          string m_name = method_h->name();
          bool found_name = false;
          for (int j = 0; j< (signed) process_sdf->size(); j++)
              {
            edges* edge_ptr = (*process_sdf)[j];
            if (strstr(method_h->name(), edge_ptr->name.c_str())
                != NULL ) {
              found_name = true;
              p_method_h = method_h;
            }/* END IF */
          }/* END FOR */
          // Check to see if the name of the current process is
              in the SDF list and route
            // accordingly to the correct METHODs list.
          if (found_name == true) {
            process_sdf_graph->sdf_method_handles.push_back(
                p_method_h);
            found_name = false;
          }
          else {
            // This must be a DE process - will be inserted
                itself
          }/* END IF-ELSE */
        }/* END IF */
      }/* END FOR */
      // remove the method handles added to SDF list from DE
          list
      for (int k = 0; k < (signed) process_sdf_graph->
          sdf_method_handles.size() ; k++) {
        sc_method_handle del_method_h = process_sdf_graph->
            sdf_method_handles[k];
        m_process_table->remove(del_method_h)->name();
      }/* END FOR */
    }
  } else {
    schedule::err_msg("NO SDF GRAPHS TO SPLIT PROCESS", "WW");
  }/* END IF-ELSE */
}/* END split_processes */
```

This implies that execution of SDF processes is performed from the DE kernel making the DE kernel the parent kernel supporting the offspring SDF kernel. We employ the parent-offspring terminology to suggest that SystemC is an event-driven modeling and simulation framework through which we establish the SDF kernel. However, this does not

mean that a DE model can not be present within an SDF model, though careful programming is required in ensuring the DE block is contained within one SDF block. For the future extension we are working on a more generic design for hierarchical kernels through an evolved API. Algorithm 5.4 displays the altered DE. The noticeable change in the kernel is the separation of initialization roles. We find it necessary to separate what we consider two distinct initialization roles as:

- Preparing model for simulation in terms of instantiating objects, setting flags, etc.
- Pushing processes (all types) onto the *runnable* queues and executing *crunch()* (see Chapter 3) once.

If there are manipulations required to the runnable process lists prior to inserting all the processes onto the *runnable* queues for execution during initialization, then the separation in initialization is necessary. For example, for the SDF kernel we require the processes that are SDF methods to be separated and not available to be pushed onto the runnable queues. This separation is performed using the *split_processes()* in the API class as shown in Figure 5.13, that identifies SDF methods and removes them from the list that holds all *SC_METHOD()* processes. This is not possible if the original initialization function for the DE kernel is unchanged because it makes all processes runnable during initialization. We are fully aware of the implications of this implementation in that it departs from the SystemC standard. However, the SystemC standard does not dictate how a Synchronous Data Flow kernel or programming paradigm is to behave, hence we feel comfortable in implementing such changes as long as the DE kernel concurs with SystemC standards. Another difference of the standard is allowing the designer to specify the order of execution of processes through an overloaded *sc_start(...)* function call. If the user has prior knowledge of a certain order in which the system, especially in the case of DE and SDF heterogeneous models, then the user should have flexibility in allowing for definition of order instead of using control signals to force order. The limitations of the SystemC standard will progressively become apparent once more MoCs are implemented resulting in more dissonance between SystemC standards and the goal of a heterogeneous modeling and simulation framework.

In addition to the separation of *SC_METHOD()* processes, the *crunch()* function that executes all the processes is slightly altered in execution of the processes. If a modeler specifies the order in which to fire the processes, then this order needs to be followed by the kernel. This requires popping all the processes from its respective runnable list

and storing them separately onto a temporary queue. These processes are then selected from this temporary queue and executed according to an order if it is specified, after which the remaining processes are executed. The need for this is to enable support for signal updates and *next_trigger()* to function correctly. When a process is sensitive to a signal and an event occurs on the signal, then this process can be *ready-to-run* causing it to be pushed onto the runnable list. If there is no ordering specified by the designer then the processes are popped regularly without requiring a temporary queue.

Likewise, the *simulate()* function suffered some alterations to incorporate one period of execution in a cycle. The *simulate()* function uses a *clock_count* variable to monitor the edges of the clock. This is necessary to enforce the SDF graph is executed once every cycle. To understand the need for it to execute once every cycle, we have to understand how an *sc_clock* object is instantiated. A clock has a positive and a negative edge and for the kernel the *sc_clock* creates two *SC_METHOD()* processes one with a positive edge and the second with a negative edge. These are then processed like normal *SC_METHOD()* processes, causing the *crunch()* function to be invoked twice. Therefore, the *clock_count* variable ensures that the SDF execution is only invoked once per cycle as per definition of a period. However, there is an interesting problem that this might result in when modeling in SDF. If there is to be a standalone SDF model, then there has to be an instance of *sc_clock* even if it is not connected to any ports in the SDF model. This is essential for the SDF graph to function correctly. We envision the SDF to be executed alongside with DE modules, hence it does not seem unnatural to expect this condition.

5.1 Specifying Execution Order for Processes

Providing the kernel with an execution order has to be carefully used by the modeler. This is because the semantics that belong to each process category (*SC_METHOD(), SC_THREAD(), SC_CTHREAD()*) are still followed. Hence, the blocks that trigger the SDF also adhere to the semantics, which can complicate execution when specifying order. Suppose an *SC_CTHREAD()*'s responsibility is to fire an SDF block and these are the only two blocks in the system such that they are called CTHREAD1 and SDF1. If the modeler specified the execution order as *"SDF1 CTHREAD1"* then the correct behavior will involve an execution of SDF1 followed by CTHREAD1 and then again SDF1. As expected, with flexibility comes complexity, but we believe allowing this kind of flexibility is necessary for modeler who understand how their model works.

5.2 SDF Kernel

The pseudo-code for the algorithm employed in altering the DE kernel to accept the SDF kernel is shown below.

Algorithm 5.4: DE Kernel and SDF Kernel
{classes edges, sdf_graph, schedule and sc_domains are already defined}

void de_initialize1()
perform update on primitive channel registries;
prepare all THREADs and CTHREADs for simulation;
{ **END initialize1()**}

void de_initialize2()
push METHOD handle onto regular DE METHOD runnable list;
push all THREADs onto THREAD runnable list;
process delta notifications;
execute crunch() to perform Evaluate-Update once.
{ **END initialize2()**}

void simulate()
initialize1();
if (clock count mod 2 == 0) **then**
  set run_sdf to true;
end if
execute crunch() until no timed notifications or runnable processes;
increment clock count;
{ **END simulate()**}

void crunch()
while (true) **do**
  if (there is a user specified order) **then**
    pop all methods and threads off runnable list and store into temporary
    while (parsed user specified order is valid) **do**
      find process handle in temporary lists and execute;
    end while
  else
    execute all remaining processes in the temporary lists;
  end if
  { Evaluate Phase }
  execute all THREAD/CTHREAD processes;
  break when no processes left to execute;
end while
{ Update Phase}
update primitive channel registries;
increment delta counter;
process delta notifications;
{ **END crunch()**}

SDF initialization is responsible for constructing an *executable schedule* for all SDFGs in the system. If any of the SDFGs is inconsistent then the simulation stops immediately, flagging that the simulation cannot be performed. This involves creating the input matrices for the Diophantine solver and creating an *executable* SDF schedule. A user calling the *sc_start(...)* function invokes the global function that in turn uses an instance of *sc_domains* that begins the initialization process of both the DE and SDF kernels as shown in Listing 5.14. *init_domain(...)* initializes the domains that exist in SystemC (SDF and DE so far) and then begins the simulation of the DE kernel (since SDF is written such that it is within the DE kernel). The *sdf_trigger(...)* global function is to be only called from the entry function of the toplevel SDF encapsulating the entire SDF model. This ensures execution of all SDF method processes specific for that SDFG.

Listing 5.14. Global functions

```
1 void sc_start( const sc_time& duration, string in )
2 {
3     model.init_domains(duration, in );
4     model.de_kernel->simulate(duration);
5 }/* END sc_start */
6
7 inline void sdf_trigger(string topname) {
8
9   if (model.sdf_domain.size() > 0)  {
10    // SDFG exists
11    model.sdf_trigger(topname);
12  }/* END IF */
13  else  {
14    schedule::err_msg("No SDFGs in current model, ensure
          add_sdf_graph() called ", "EE");
15    return;
16  }/* END IF-ELSE */
17 }/* END sdf_trigger */
```

The API class *sc_domains* described in Chapter 8 has function declarations to initialize the DE and SDF kernels implemented at the API level to invoke their respective DE or SDF functions. The *init_domains(...)* function shown in Listing 5.15 is responsible for initializing all the existing domains in SystemC. This function sets the user order string if specified then prepares the simulation in terms of instantiating objects and setting the simulation flags, splits the processes as explained earlier and readies the runnable queues. Followed by initialization of the SDF, which traverses through all SDFGs present in the system and creates an *executable schedule* for each one if one can be computed. Given that all conditions are satisfied and simulation is not halted during the scheduling process, the simulation begins.

Listing 5.15. init_domain() in sc_domains

```
1// initial the domains
2void sc_domains::init_domains(const sc_time & duration, string
      in ) {
3
4  if (in.size() > 0)
5    user_input(in);
6
7  // initialize the simulation flags
8  init_de();
9  // split the processes for every SDFG
10 split_processes();
11 // initialize the runnable lists
12 model.de_kernel->de_initialize2();
13 // initialize SDF for execution
14 init_sdf();
15}
```

Listing 5.16. sdf_trigger() and *find_sdf_graph()* in sc_domains

```
1void sc_domains::sdf_trigger(string topname)  {
2
3  string sdfname = topname+".";
4  sdf_graph * run_this;
5
6  for (int sdf_graphs = 0; sdf_graphs < (signed) model.
       sdf_domain.size(); sdf_graphs++) {
7    // pointer to a particular SDF graph
8    sdf_graph * process_sdf_graph = model.sdf_domain[sdf_graphs
         ];
9    if (strcmp(process_sdf_graph->prefix.c_str(), sdfname.c_str
         ())==0 ) {
10     run_this = process_sdf_graph;
11     if ( run_sdf == true){
12       // execute the SDF METHODs
13       run_this->sdf_simulate(sdfname);
14       run_sdf = false;
15     }/* END IF */
16   }/* END IF */
17 }/* END FOR */
18}/* END sdf_trigger */
19
20sdf_graph * sc_domains::find_sdf_graph(string sdf_prefix) {
21
22 for (int sdf_graphs = 0; sdf_graphs < (signed) model.
       sdf_domain.size(); sdf_graphs++) {
23   // pointer to a particular SDF graph
24   sdf_graph * process_sdf_graph = model.sdf_domain[sdf_graphs
         ];
25   if (strcmp(process_sdf_graph->prefix.c_str(), sdf_prefix.
         c_str())==0 )  {
26     return ( process_sdf_graph);
27   }/* END IF */
28 }/* END FOR */
29   return NULL;
30}/* END find_sdf_graph */
```

Listing 5.16 shows the definition of *sdf_trigger()* that calls the *sdf_simulate()* function responsible for finding the appropriate SDFG

(with the helper function *find_sdf_graph(...)* and executing the SDF processes corresponding to that SDFG.

The creation of the schedules is encapsulated in the *sdf_create_schedule(...)* function that constructs the topology matrix for the Diophantine equations solver, returns the solution from the solver and creates an *executable schedule* if one exists as demonstrated in Listing 5.17.

Listing 5.17. SDF initialization function

```
1 void sdf_graph::sdf_create_schedule() {
2
3   // Repeat for all the SDF graphs that are modelled
4   // Extract the address of first SDF
5   sdf_graph * process_sdf_graph = this;
6   int* input_matrix = schedule::create_schedule(
        process_sdf_graph);
7
8   if (input_matrix != NULL) {
9     hbs(process_sdf_graph->num_arcs, process_sdf_graph->
          num_nodes, input_matrix,&process_sdf_graph->result);
10
11    if ( process_sdf_graph->result != NULL) {
12      // Stored the address of the schedule in the resultlist.
13    }
14    else  {
15      schedule::err_msg("The result schedule is invalid. Halt
            simulation", "EE");
16      // Attempt at clean up for Matrix Input
17      free(input_matrix);
18      // Exit simulation
19      exit(1);
20    }/* END IF-ELSE */
21    free(input_matrix);
22  }/* END IF */
23  else {
24    schedule::err_msg("Input Matrix for SDF is invalid. Halt
          simulation", "EE");
25    exit(1);
26  }/* END IF-ELSE */
27  schedule::construct_valid_schedule(*process_sdf_graph);
28}/* END sdf_create_schedule */
```

The compilation of SystemC requires passing a compiler flag *_SDF_KERNEL_ENABLE*; hence the *#ifndef*s with that variable. The first stage is in creating a matrix representation that can be the input for the linear Diophantine equation solver. This is performed by traversing through all SDFGs that might exist in the system and passing a pointer to the SDFG as an argument to the *create_schedule(...)* function that creates the topology matrix. If the pointer to the topology matrix called *input_matrix* is valid (not *NULL*) then it is passed to the Diophantine equation solver via the *hbs*(...) call, which depending on whether there exists a solution returns a pointer with the result or returns *NULL*. The address of the matrix with the result is stored in that SDFG's *re-*

sult data member. Once the result is calculated it is checked whether it is not *NULL* and if it is then the simulation exits, otherwise the simulation proceeds by executing the *construct_valid_schedule(...)* function. Remember that the schedule class is a purely static class with all its member functions being static hence all function calls precede with *schedule::*. If at any point during the schedule construction either of the input to the Diophantine solver, or the result or even the scheduling for *firing order* finds inconsistencies then the simulation exits. The method of exit might not be the safest exit using the *exit(1)* function and an improvement might involve creating a better cleanup and exit mechanism or perhaps the use of *sc_stop()*.

Upon construction of the SDF *execution schedule*s the simulation begins. When a toplevel SDF process is identified, the user is required to invoke *sdf_trigger()* from its entry function leading to a brief discussion of that function shown in Listing 5.18. The underlying function called from within *sdf_trigger* is *sdf_crunch()* that actually executes the SDF processes as shown in Listing 5.18. This function traverses through the SDF method processes stored in this particular SDFG and executes them in the order created by the scheduling algorithm. It must be noted that these SDF processes are not pushed onto the runnable queues restricting their functionality by not allowing a safe usage of *next_trigger()* or the sensitivity lists.

The *sdf_simulate(...)* function simply calls the *sdf_crunch(...)* function as in Listing 5.19. The full source with changes in the original kernel to incorporate the SDF kernel is available at [36]. We have only discussed the main functions that are used to invoke the SDF kernel and point out to the reader to the few changes made to the original DE kernel to accommodate the SDF kernel. To summarize, we list the changes made to the existing kernel:

- Separation of initialization roles to allow for separation of processes.
- Splitting of *SC_METHOD()* processes. Not all *SC_METHOD()* processes are made runnable since we remove SDF block methods from the method process list.
- Specifying user order handles processes of all types. If a user order is specified then a temporary queue is used to pop the runnable lists, find the appropriate processes and execute them.
- Addition of a clock counter to invoke the SDF execution only once per cycle when the toplevel entry function is fired.

We believe these to be moderate changes given that we implement a completely new kernel working alongside with the DE kernel and accept

Listing 5.18. SDF crunch function

```
1 void sdf_graph::sdf_crunch(string sdf_prefix) {
2
3   sc_method_handle method_h;
4   #ifdef _SDF_KERNEL_ENABLE
5   /* Executes only the SDF methods that have been initialized.
        Ordinary DE based
6   METHODs will be executed later.
7   */
8   sdf_graph * process_this_sdf_graph = this; //find_sdf_graph(
        sdf_prefix);
9
10  if ( process_this_sdf_graph != NULL) {
11  // pointer to particular edges* list
12    sdf * process_sdf = &process_this_sdf_graph->sdf_schedule;
13
14    // traverse through all the scheduled nodes in that
          particular order
15    for (int j = 0; j < (signed) process_sdf->size(); j++)  {
16      edges* edg_ptr = (*process_sdf)[j];
17      for (int i = 0; i < (signed) process_this_sdf_graph->
            sdf_method_handles.size(); i++)  { //
            asdfasdfasfdasdfasd
18        method_h = process_this_sdf_graph->sdf_method_handles[i
            ];
19        if (strstr(method_h->name(), edg_ptr->name.c_str()))  {
20          try    {
21                method_h->execute();
22          }
23          catch( const sc_exception& ex )    {
24               cout << "\n" << ex.what() << endl;
25               return;
26          }
27        }/* END IF */
28      }/* END FOR */
29    }/* END FOR */
30  }
31  else {
32    schedule::err_msg("SDF Graph not found ", "EE");
33  }/* END IF-ELSE */
34  #endif
35}/* END sdf_crunch */
```

Listing 5.19. SDF simulate function

```
1 void sc_simcontext::simulate( const sc_time& duration ) {
2   // Execute for a period (one execution of SDF)
3   sdf_crunch(prefix);
4 }
```

that there can be better conceivable methods of implementing this as the C++ language allows an infinite number of implementations.

6. SDF Specific Examples

Experimentation of the SDF kernel has been an incremental process beginning with pure SDF models followed by heterogeneous models. To

evaluate the efficiency enhancement by our SDF kernel we set out to experiment with a few models that are amenable to SDF style modeling. In this section, we show the results of simulating the same models for three distinct modeling styles and different sample sizes of data. The first set of experiments are pure SDF models for the Finite Impulse Response (FIR), Fast Fourier Transform (FFT), and the Sobel edge detection algorithm [47]. The second set involves creating a combination of Discrete-Event and Synchronous Data Flow models shown in Chapter 9. In [62] around 50% or more improvement in simulation efficiency over threaded models have been reported. Our aim has been to improve upon those results reported in [62], which we call "Non-Threaded" models. Furthermore, we aspire in betterment of the modeling paradigm for SDF-like systems.

7. Pure SDF Examples

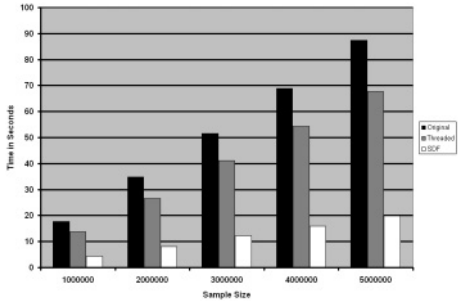
Three systems are modeled using the SDF kernel and the SDF modeling style. They are Finite Impulse Response Filter (FIR), Fast Fourier Transform (FFT), and a Sobel edge detection system. The same systems are modeled with the original standard SystemC DE kernel and the data is shown in Figures 5.6. These experiments are executed on a Linux 2.4.18 platform with an *Intel*® *Pentium*® IV CPU 2.00GHz processor, 512KB cache size, and 512MB of RAM. The first two are taken from SystemC examples distribution.

We already discussed the FIR model in earlier chapters allowing us to move on to the next example, which is the FFT model. The FFT model is also a three-staged model with a Source block, FFT block and a Sink block having the responsibilities of creating the input, performing FFT on the input, and displaying the result, respectively. Figure 5.7 shows the block diagram describing this model.

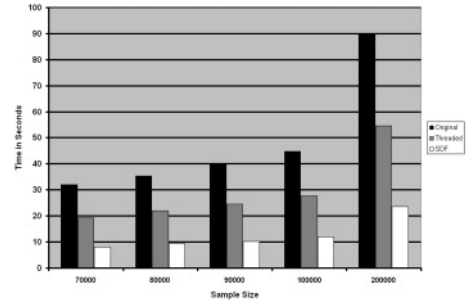
The third example chosen is the Sobel example [53, 62]. This example is a five-staged model shown in Figure 5.8. The Input block simply reads the matrix from an input file and pushes the values into the data path queue. This queue is an input to the CleanEdges block that clears the edges of the matrix and pushes the values through the Channel block to the Sobel operator block. This Sobel block performs the Sobel computations and pushes the results to the Output block completing the entire edge detection example.

We construct these three models with the following underlying scenarios:

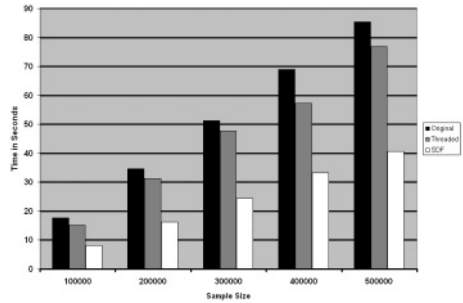
- Discrete-Event kernel using *SC_THREAD()/ SC_CTHREAD()* processes for modeling.



(a) FIR



(b) FFT



(c) Sobel

(Black = Original DE models, Dark Gray = Non-Threaded models, White = SDF models)

Figure 5.6. Results from Experiments

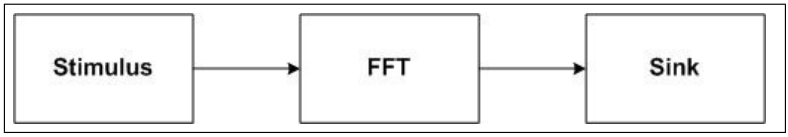


Figure 5.7. FFT Block Diagram

- Discrete-Event kernel with the non-threaded models (transformed using the transformation technique specified in [62]).
- Synchronous Data Flow implementation using the implemented SDF kernel.

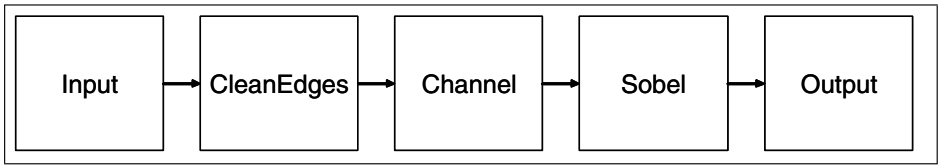


Figure 5.8. Sobel Block Diagram

The graphs in Figure 5.6 present results from these three scenarios. They show that every model demonstrates significant improvement in the amount of simulation time over the original model and the non-threaded model. The three bars on each chart refer to the time taken in seconds for the entire model to be executed in their respective modeling scenarios. The leftmost bar being the original, middle bar being non-threaded are modeled using the DE kernel and the rightmost bar using the SDF kernel. The bar charts show that increasing the number of sample size will still preserve the efficiency presented by the set of collected data. The FIR and FFT yielded approximately 75% improvement in simulation time compared to the original model and the Sobel yielded a 53% improvement in simulation time. Comparing results from the SDF kernel to the non-threaded models, the FIR, FFT and Sobel, showed 70%, 57% and 47% improvement in simulation time respectively. All results show better performance with the SDF kernel than both the original and non-threaded models. We conducted an investigation on the relative lower improvement for the Sobel model and understand that the Input block in Sobel is only executed once to read in the entire matrix of values. However, when this is altered to select segments of the matrix then the performance reflects the FIR and FFT model results. This indicates that simulation efficiency depends on the number of invocations of the blocks required to perform certain functions. The Sobel model using the DE kernel required a lot more invocations when collecting segments of the matrix increasing the signal communication and data passing. Future experimentation proposes comparison with different matrix segment sizes for this edge detection algorithm. The percentage improvement over the non-threaded model in [62] is also consistent with the Sobel edge detection yielding a lower improvement and for the same reason.